The Overture Grid Classes
Reference Guide, Version 1.0

Geoffrey S. Chesshire

Scientific Computing Group (CIC-19)
Los Alamos National Laboratory
Los Alamos, New Mexico 87545, USA

William D. Henshaw

Centre for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA, 94551
henshaw@llnl.gov
http://www.llnl.gov/casc/people/henshaw
http://www.llnl.gov/casc/Overture August 7, 2002 UCRL-MA-134448

Abstract

Overture is a library containing classes for grids, overlapping grid generation and the discretization and solution of PDEs on overlapping grids. This document describes the Overture grid classes, including classes for single grids and classes for collections of grids. The primary classes described are the **MappedGrid**, **GridCollection** and **CompositeGrid** classes. These classes hold the geometry arrays required by PDE solvers such as the **vertex** (grid vertices), **vertexDerivative** (jacobian derivatives), and **vertexBoundaryNormal** (normals on the boundary), etc. The geometry arrays can be optionally generated as required by the application. The grid classes have support for multigrid levels and for adaptive mesh refinement.

Contents

1 Introduction

The Overture grid classes include classes for single grids and classes for collections of grids. The single-grid classes are related to each other through the C++ inheritance mechanism. The base class is **GenericGrid** (§2), and the class **MappedGrid** (§3) is derived from **GenericGrid**. The collections of grids are also related to each other through inheritance. The base class for collections of grids is **GenericGridCollection** (§4), which contains a list of **GenericGrid**s. The class **GridCollection** (§5) is derived from GenericGridCollection, and contains a list of **MappedGrid**s. All other Overture grid classes that contain collections of **MappedGrid**s are derived from **GridCollection**. In particular, the class **CompositeGrid** (§6) is derived from **GridCollection**. All of these classes are described in this document.

All of the Overture grid classes are reference-counted, using the envelope-letter paradigm. To support this, the base grid classes **GenericGrid** and **GenericGridCollection** are derived from the base class **ReferenceCounting** (Appendix A).

2 Class GenericGrid

Note: You should not need to read this section unless you are designing a derived grid class.

Class **GenericGrid** is the base class for all of the Overture single-grid classes. By itself it does not contain any geometric data. It is useful only as a base class for other grid classes that may contain data to describe particular kinds of grids. We envision deriving from **GenericGrid** separate classes for structured and unstructured grids, and perhaps for other kinds of grids that we have not anticipated. For example, the class **MappedGrid** (§3) is derived from **GenericGrid** in order to describe curvilinear structured grids.

Many of the public constants, member data and member functions of class **GenericGrid** are overloaded in the derived classes. They are defined here in the base class for single grids because they are common to all single grid classes. The ordinary user (programmer) need not be concerned with these constants, data and member functions, or with the class **GenericGrid** at all, except where they are explicitly referred to in the descriptions of derived grid classes such as **MappedGrid** (§3).

2.1 Public member functions

2.1.1 GenericGrid()

Default constructor.

2.1.2 GenericGrid(const GenericGrid& x, const CopyType ct = DEEP)

Copy constructor. This does a deep copy by default. See also **operator=(x)** (§2.1.4) and **reference(x)** (§2.1.5).

2.1.3 virtual ~GenericGrid()

Destructor.

2.1.4 GenericGrid& operator=(const GenericGrid& x)

Assignment operator. This is also called a deep copy.

2.1.5 void reference(const GenericGrid& x)

Make a reference. This is also called a shallow copy. This **GenericGrid** shares the data of **x**.

2.1.6 virtual void breakReference()

Break a reference. If this **GenericGrid** shares data with any other **GenericGrid**, then this function replaces it with a new copy that does not share data.

2.1.7 virtual void consistencyCheck() const

Check the consistency of this **GenericGrid**.

2.1.8 virtual Integer get(const GenericDataBase& dir, const aString& name)

Copy a **GenericGrid** from a file.

2.1.9 virtual Integer put(GenericDataBase& dir, const aString& name) const

Copy a **GenericGrid** into a file.

2.1.10 Integer update(const Integer what = THEusualSuspects, const Integer how = COMPUTEtheUsual)

Update geometric data. The first argument (**what**) indicates which geometric data are to be updated. Any combination of the constants **NOTHING** (§2.5.1), **THEusualSuspects** (§2.5.2) and **EVERYTHING** (§2.5.3) may be bitwise ORed together to form the first argument of **update**(), to indicate which geometric data should be updated. This function returns a value obtained by bitwise ORing some of these constants, to indicate for which of the optional geometric data new array space was allocated. In addition, the constant **COMPUTEfailed** (§2.5.6), may be bitwise ORed into the value returned by **update**() in order to indicate that the computation of some geometric data failed. The second argument (**how**) indicates whether and how any computation of geometric data should be done. Any combination of the constants **COMPUTEnothing** (§2.5.4) and **COMPUTEtheUsual** (§2.5.5) may be bitwise ORed together to form the optional second argument of **update**(). In fact, a **GenericGrid** contains no geometric data, so all this is irrelevant.

2.1.11 virtual Integer update(GenericGrid& x, const Integer what = THEusualSuspects, const Integer how = COMPUTEtheUsual)

Update geometric data, sharing space with the optional geometric data of another **GenericGrid** (**x**). If space for any indicated optional geometric data has not yet been allocated, or has the wrong dimensions, but **x** does contain the corresponding data, then the data for this **GenericGrid** will share space with the corresponding data of **x**. Any geometric data that already exists and has the correct dimensions is not forced to share space with the corresponding data of **x**. For the optional arguments **what** and **how**, see the description of the function **update(what,how)** (§2.1.10).

2.1.12 virtual void destroy(const Integer what = NOTHING)

Destroy the indicated optional **GenericGrid** geometric data. The argument (**what**) indicates which optional geometric data are to be destroyed. Any combination of the constants **NOTHING** (§2.5.1), **THEusualSuspects** (§2.5.2) and **EVERYTHING** (§2.5.3) may be bitwise ORed together to form the optional argument **what**.

2.1.13 void geometryHasChanged(const Integer what = ∼NOTHING)

Mark the geometric data out-of-date. Any combination of the constants **NOTHING** (§2.5.1), **THEusualSuspects** (§2.5.2) and **EVERYTHING** (§2.5.3) may be bitwise ORed together to form the first argument of **geometryHasChanged**(). By default, all geometric data of this **GenericGrid** and all derived classes is marked out-of-date. It is recommended that this function be called only from derived classes and grid-generation programs.

2.1.14 Logical operator==(const GenericGrid& x) const

This comparison function returns **LogicalTrue** (non-zero) if and only if **x** refers the same grid as ∗**this**.

2.1.15 Logical operator!=(const GenericGrid& x) const

This comparison function returns **LogicalTrue** or (non-zero) if and only if **x** does not refer to the same grid as ∗**this**.

2.2 Public Member functions for access to data

2.2.1 const Integer& computedGeometry() const

This function returns a reference to a bit mask that indicates which geometrical data has been computed. This mask must be reset to zero to invalidate the data when the geometry changes. It is recommended that this data be used only by derived classes and grid-generation programs. See also **geometryHasChanged(what)** (§2.1.13).

2.2.2 virtual aString getClassName() const

Get the class name of the most-derived class for this object.

2.2.3 GenericGridData* operator->()

Access the reference-counted data.

2.2.4 GenericGridData& operator*()

Access the pointer to the reference-counted data.

2.3 Public member functions called only from derived classes

It is recommended that these functions be called only from derived classes.

2.3.1 void reference(GenericGridData& x)

Make a reference to an object of class **GenericGridData**. This **GenericGrid** uses **x** for its data. It is recommended that this function be called only from derived classes.

2.3.2 void updateReferences(const Integer what = EVERYTHING)

Update references to the reference-counted data. It is recommended that this function be called only from derived classes.

2.4 Public data used only by derived classes

It is recommended that these variables be used only by derived classes.

2.4.1 GenericGridData* rcData

rcData is a pointer to the reference-counted data. It is recommended that this variable be used only by derived classes. See also the member functions **operator->()** (§2.2.3) and **operator*()** (§2.2.4), which are provided for access to **rcData**.

2.4.2 Logical isCounted

isCounted is a flag that indicates whether the data pointed to by **rcData** (§2.4.1) is known to be reference-counted. It is recommended that this variable be used only by derived classes.

2.5 Public constants

2.5.1 NOTHING

NOTHING = 0
NOTHING indicates no geometric data. See also **update(what,how)** (§2.1.10) and **destroy(what)** (§2.1.12).

2.5.2 THEusualSuspects

THEusualSuspects = **NOTHING** (§2.5.1)
THEusualSuspects indicates some of the geometric data of a **GenericGrid**. The particular data indicated by **THEusualSuspects** may change from time to time. For this reason the use of **THEusualSuspects** is not recommended. In fact, a **GenericGrid** contains no geometric data, so all this is moot. This constant is typically overloaded in a derived class, to indicate some of the geometric data of that class, in addition to the geometric data indicated by the constant **THEusualSuspects** defined in its base class. See also **update(what,how)** (§2.1.10) and **destroy(what)** (§2.1.12).

2.5.3 EVERYTHING

EVERYTHING = **NOTHING** (§2.5.1)
EVERYTHING indicates all of the geometric data associated with a **GenericGrid**. In fact, a **GenericGrid** contains no geometric data. This constant is typically overloaded in a derived class, to indicate all of the geometric data of that class, in addition to the geometric data indicated by the constant **EVERYTHING** defined in its base class. See also **update(what,how)** (§2.1.10) and **destroy(what)** (§2.1.12).

2.5.4 COMPUTEnothing

COMPUTEnothing = 0
COMPUTEnotning indicates that no geometric data should be computed. See also **update(what,how)** (§2.1.10).

2.5.5 COMPUTEtheUsual

COMPUTEtheUsual = **COMPUTEnothing** (§2.5.4)
COMPUTEtheUsual indicates that computation of geometric data should proceed in the "usual way." In fact, a **GenericGrid** contains no geometric data, so this is irrelevant. This constant is typically overloaded in a derived class, to indicate the "usual way" of computing geometry relevant to that class, in addition to the usual way of computing the geometric data indicated by the constant **COMPUTEtheUsual** defined in and relevant to its base class. See also **update(what,how)** (§2.1.10).

2.5.6 COMPUTEfailed

COMPUTEfailed indicates that computation of some geometric data failed. See also **update(what,how)** (§2.1.10).

3 Class MappedGrid

Class **MappedGrid** is used for all logically-rectangular grids. This includes cartesian, rectangular and curvilinear grids. Class MappedGrid allows for grids with holes, unused vertices or cells within a grid. It is assumed that a continuous function exists which maps the vertices of a uniform grid to the vertices of the MappedGrid. This is no restriction, becauase it is always possible to construct a function, for example an interpolant, with this property.

Class **MappedGrid** is derived from class **GenericGrid** (§2). It overloads some of the **GenericGrid** public constants, member data and member functions.

3.1 Public member functions

3.1.1 MappedGrid(const Integer numberOfDimensions_ = 0)

Default constructor. If numberOfDimensions_==0 (*e.g.*, by default) then create a null **MappedGrid**. Otherwise, create a **MappedGrid** with the given number of dimensions.

3.1.2 MappedGrid(const MappedGrid& x, const CopyType ct = DEEP)

Copy constructor. This does a deep copy by default. See also **operator=(x)** (§3.1.6) and **reference(const MappedGrid& x)** (§3.1.7).

3.1.3 MappedGrid(Mapping& mapping_)

Constructor from a mapping.

3.1.4 MappedGrid(MappingRC& mapping_)

Constructor from a reference-counted mapping.

3.1.5 virtual ~MappedGrid()

Destructor.

3.1.6 MappedGrid& operator=(const MappedGrid& x)

Assignment operator. This is also called a deep copy.

3.1.7 void reference(const MappedGrid& x)

Make a reference. This is also called a shallow copy. This **MappedGrid** shares the data of **x**.

3.1.8 void reference(Mapping& x)

Use a given mapping.

3.1.9 void reference(MappingRC& x)

Use a given reference-counted mapping.

3.1.10 virtual void breakReference()

Break a reference. If this **MappedGrid** shares data with any other **MappedGrid**, then this function replaces it with a new copy that does not share data.

3.1.11 void changeToAllVertexCentered()

Change the grid to be all vertex-centered.

3.1.12 void changeToAllCellCentered()

Change the grid to be all cell-centered.

3.1.13 virtual void consistencyCheck() const

Check the consistency of this **MappedGrid**.

3.1.14 virtual Integer get(const GenericDataBase& dir, const aString& name)

Copy a **MappedGrid** from a file.

3.1.15 virtual Integer put(GenericDataBase& dir, const aString& name) const

Copy a **MappedGrid** into a file.

3.1.16 Integer update(const Integer what = THEusualSuspects, const Integer how = COMPUTEtheUsual)

Update geometric data. The first argument (**what**) indicates which geometric data are to be updated. Any combination of the constants **THEmask** (§3.6.1), **THEvertex** (§3.6.2), **THEcenter** (§3.6.3), **THEcorner** (§3.6.4), **THEvertexDerivative** (§3.6.5), **THEcenterDerivative** (§3.6.6), **THEinverseVertexDerivative** (§3.6.7), **THEinverseCenterDerivative** (§3.6.8), **THEvertexJacobian** (§3.6.9), **THEcenterJacobian** (§3.6.10), **THEcellVolume** (§3.6.11), **THEfaceNormal** (§3.6.12), **THEcenterNormal** (§3.6.13), **THEfaceArea** (§3.6.14), **THEcenterArea** (§3.6.15), **THEvertexBoundaryNormal** (§3.6.16), **THEcenterBoundaryNormal** (§3.6.17), **THEcenterBoundaryTangent** (§3.6.18), **THEminMaxEdgeLength** (§3.6.19), **THEboundingBox** (§3.6.20), **THEusualSuspects** (§3.6.21) and **EVERYTHING** (§3.6.22), as well as any of the corresponding constants allowed for **GenericGrid::update(what,how)** (§2.1.10), may be bitwise ORed together to form the first argument of **update**(), to indicate which geometric data should be updated. This function returns a value obtained by bitwise ORing some of these constants, to indicate for which of the optional geometric data new array space was allocated. In addition, the constant **COMPUTEfailed** (§2.5.6), may be bitwise ORed into the value returned by **update**() in order to indicate that the computation of some geometric data failed. The second argument (**how**) indicates whether and how any computation of geometric data should be done. Any combination of the constants **USEdifferenceApproximation** (§3.6.23), **COMPUTEgeometry** (§3.6.24), **COMPUTEgeometryAsNeeded** (§3.6.25), **COMPUTEtheUsual** (§3.6.26), as well as any of the corresponding constants allowed for **GenericGrid::update(what,how)** (§2.1.10), may be bitwise ORed together to form the optional second argument of **update**(). The corresponding function **update(what,how)** (§2.1.10) is called with the same arguments for the base class **GenericGrid**.

3.1.17 Integer update(MappedGrid& x, const Integer what = THEusualSuspects, const Integer how = COMPUTEtheUsual)

Update geometric data, sharing space with the optional geometric data of another grid (**x**). If space for any indicated optional geometric data has not yet been allocated, or has the wrong dimensions, but **x** does contain the corresponding data, then the data for this **MappedGrid** will share space with the corresponding data of **x**. Any geometric data that already exists and has the correct dimensions is not forced to share space with the corresponding data of **x**. The corresponding function **update(x,what,how)** (§2.1.11) is called with the same arguments for the base class **GenericGrid**. For the optional arguments **what** and **how**, see the description of the function **update(what,how)** (§3.1.16).

3.1.18 void destroy(const Integer what = NOTHING)

Destroy the indicated optional geometric grid data. The argument (**what**) indicates which optional geometric data are to be destroyed. Any combination of the constants **THEmask** (§3.6.1), **THEvertex** (§3.6.2), **THEcenter** (§3.6.3), **THEcorner** (§3.6.4), **THEvertexDerivative** (§3.6.5), **THEcenterDerivative** (§3.6.6), **THEinverseVertexDerivative** (§3.6.7), **THEinverseCenterDerivative** (§3.6.8), **THEvertexJacobian** (§3.6.9), **THEcenterJacobian** (§3.6.10), **THEcellVolume** (§3.6.11), **THEfaceNormal** (§3.6.12), **THEcenterNormal** (§3.6.13), **THEfaceArea** (§3.6.14), **THEcenterArea** (§3.6.15), **THEvertexBoundaryNormal** (§3.6.16), **THEcenterBoundaryNormal** (§3.6.17), **THEcenterBoundaryTangent** (§3.6.18), **THEminMaxEdgeLength** (§3.6.19), **THEboundingBox** (§3.6.20), **THEusualSuspects** (§3.6.21) and **EVERYTHING** (§3.6.22), as well as any of the corresponding constants allowed for **GenericGrid::destroy(what)** (§2.1.12), may be bitwise ORed together to form the optional argument **what**. The corresponding function **destroy(what)** (§2.1.12) is called with the same argument for the base class **GenericGrid**.

3.1.19 void getInverseCondition(MappedGrid& g2, const RealArray& xr1, const RealArray& rx2, const RealArray& condition)

g2 (INPUT) The **MappedGrid** whose inverse mapping was used to compute **rx2**.

xr1 (INPUT) Dimensions: $(N, 0{:}2, 0{:}2)$
The derivative of the mapping of this **MappedGrid**. The range of points N for which the inverse condition is computed is determined by the first dimension of **xr1**. The first dimension of **rx2** and **condition** should be the same as the first the dimension of **xr1**.

rx2 (INPUT) Dimensions: $(N, 0{:}2, 0{:}2)$
The inverse derivative of the mapping of **MappedGrid g2**.

condition (OUTPUT) Dimensions: (N)
The condition number of the inverse.

This function computes the condition number of the mapping inverse.

$$\textbf{condition}(i) = \left\| \operatorname{diag}\left(1/\textbf{g2.gridSpacing}(*)\right)\left[\textbf{rx2}(i,*,*)\right]\left[\textbf{xr1}(i,*,*)\right]\operatorname{diag}\left(\textbf{gridSpacing}(*)\right)\right\|_\infty$$

3.1.20 void specifyProcesses(const Range& range)

Specify the set of processes over which **MappedGridFunction**s are distributed. We now support only the specification of a contiguous range of virtual process IDs.

3.1.21 virtual void initialize(const Integer& numberOfDimensions_)

Initialize the MappedGrid with the given number of dimensions. The number of dimensions given must be consistent with the number of dimensions of the **Mapping** used by the grid.

3.2 Public Member functions for access to data

3.2.1 const Integer& numberOfDimensions() const

This function returns a reference to the number of dimensions of the domain.

3.2.2 const IntegerArray& dimension() const

Dimensions: $(0{:}1, 0{:}2)$
dimension() holds the dimensions (lower and upper index bounds) for the indices corresponding to coordinates in the parameter space of the grid, of **MappedGridFunctions** of all types defined on the grid. **dimension**(i,j) refers to the side of the grid corresponding to the coordinate value $r_j = i$ in the parameter space of the grid. If the grid has ghost points, then **dimension**$() \neq$ **gridIndexRange**$()$. For the extra dimensions **numberOfDimensions**$() \leq j \leq 2$, if any, **dimension**$(i,j) = 0$. This data is always recomputed by the function **update(what,how)** (§3.1.16).

3.2.3 const IntegerArray& indexRange() const

Dimensions: $(0{:}1, 0{:}2)$
indexRange() holds the range of indices of the discretization points. **indexRange**(i,j) refers to the side of the grid corresponding to the coordinate value $r_j = i$ in the parameter space of the grid. In those coordinate directions j where the grid is neither cell-centered nor periodic, and for $j \geq$ **numberOfDimensions**$()$, the discretization points have the same index range as the vertices of the grid, so **indexRange**$(i,j) =$ **gridIndexRange**(i,j) for $i = 0$ and for $i = 1$. In cell-centered or periodic coordinate directions j, the first discretization point has the same index as the first vertex, so **indexRange**$(0,j) =$ **gridIndexRange**$(0,j)$, but there is one discretization point fewer than the number of vertices, so **indexRange**$(1,j) =$ **gridIndexRange**$(1,j) - 1$. For the extra dimensions **numberOfDimensions**$() \leq j \leq 2$, if any, **indexRange**$(i,j) = 0$. This data is always recomputed by the function **update(what,how)** (§3.1.16).

3.2.4 const IntegerArray& extendedIndexRange() const

Dimensions: $(0{:}1, 0{:}2)$
extendedIndexRange() holds the range of indices of the discretization points and interpolation points. **extendedIndexRange**(i,j) refers to the side of the grid corresponding to the coordinate value $r_j = i$ in the parameter space of the grid. On those sides of the grid (i,j) where the grid has a non-zero boundary condition, or where the grid is periodic in coordinate direction j, and for $j \geq$ **numberOfDimensions**$()$, the interpolation points have the same index range as the discretization points, so **extendedIndexRange**$(i,j) =$ **indexRange**(i,j). On those sides of the grid (i,j) where the grid has a zero boundary condition, the index range of the interpolation points extends outside the index range of the discretization points by the lesser of $($**discretizationWidth**$(j) - 1)/2$ and **numberOfGhostPoints**(i,j). For the extra dimensions **numberOfDimensions**$() \leq j \leq 2$, if any, **extendedIndexRange**$(i,j) = 0$. This data is always recomputed by the function **update(what,how)** (§3.1.16).

3.2.5 const IntegerArray& gridIndexRange() const

Dimensions: $(0{:}1, 0{:}2)$
gridIndexRange() holds the range of indices of the grid vertices. **gridIndexRange**(i,j) refers to the side of the grid corresponding to the coordinate value $r_j = i$ in the parameter space of the grid. For the extra dimensions **numberOfDimensions**$() \leq j \leq 2$, if any, **gridIndexRange**$(i,j) = 0$. This data may be set using the function **setGridIndexRange(ks,kd,gridIndexRange_)** (§3.2.6).

3.2.6 void setGridIndexRange(const Integer& ks, const Integer& kd, const Integer& gridIndexRange_)

This function is used to change the value of **gridIndexRange()** (§3.2.5). If the new value is different from the old value, then all geometric data is destroyed.

3.2.7 const IntegerArray& numberOfGhostPoints() const

Dimensions: $(0{:}1, 0{:}2)$
numberOfGhostPoints() holds the number of ghost point vertices on each side of the grid. **numberOfGhostPoints**(i,j) refers to the side of the grid corresponding to the coordinate value $r_j = i$ in the parameter space of the grid. The number of ghost points is the difference between the corresponding bounds on the grid vertex index range and on the dimensions, so

$$\textbf{numberOfGhostPoints}(i,j) = (-1)^i\big(\textbf{gridIndexRange}(i,j) - \textbf{dimension}(i,j)\big).$$

This data may be set using the function **setNumberOfGhostPoints(ks,kd,numberOfGhostPoints_)** (§3.2.8).

3.2.8 void setNumberOfGhostPoints(const Integer& ks, const Integer& kd, const Integer& numberOfGhostPoints_)

This function is used to change the value of **numberOfGhostPoints()** (§3.2.7). If the new value is different from the old value, then all geometric data is destroyed.

3.2.9 const Logical& useGhostPoints() const

This function returns a reference to a flag that is **LogicalTrue** (non-zero) if and only if ghost points on the grid are used on boundaries where the boundary condition is zero. This data may be set using the function **setUseGhostPoints(useGhostPoints_)** (§3.2.10).

3.2.10 void setUseGhostPoints(const Logical& useGhostPoints_)

This function is used to change the value of **useGhostPoints()** (§3.2.9). If the new value is different from the old value, then all geometric data is destroyed.

3.2.11 const IntegerArray& discretizationWidth() const

Dimensions: $(0{:}2)$
discretizationWidth() holds the width of the interior discretization stencil. This means that every interior discretization point is guaranteed to have available to it a stencil of this width consisting of valid points for use in the discretization of a PDE. Points that are so close to the boundary that such a stencil would extend outside the grid are not considered to be interior discretization points, but may be boundary discretization points. **discretizationWidth**(i) refers to the width of the stencil in the direction corresponding to the coordinate r_i in the parameter space of the grid. For the extra dimensions **numberOfDimensions**$() \le i \le 2$, if any, **discretizationWidth**$(i) = 0$. This data may be set using the function **setDiscretizationWidth(kd,discretizationWidth_)** (§3.2.12).

3.2.12 void setDiscretizationWidth(const Integer& kd, const Integer& discretizationWidth_)

This function is used to change the value of **discretizationWidth()** (§3.2.11).

3.2.13 const IntegerArray& boundaryDiscretizationWidth() const

Dimensions: $(0{:}1, 0{:}2)$
boundaryDiscretizationWidth() holds the width of the boundary condition discretization stencil in the direction normal to the boundary, on each side of the grid. This means that every boundary discretization point is guaranteed to have available to it a one-sided stencil of this width consisting of valid points for use in the discretization of boundary conditions. This stencil includes points on the boundary and extends from there into the interior of the grid. **boundaryDiscretizationWidth**(i,j) refers to the side of the grid corresponding to the coordinate value $r_j = i$ in the parameter space of the grid. The boundary condition stencil width does not consider any ghost points. It considers only points on the boundary and inside the grid. In addition, any number of ghost points may be used as needed for the discretization of boundary conditions. For the extra dimensions **numberOfDimensions**$() \le j \le 2$, if any, **boundaryDiscretizationWidth**$(i,j) = 0$. This data may be set using the function **setBoundaryDiscretizationWidth(ks,kd,boundaryDiscretizationWidth_)** (§3.2.14).

3.2.14 void setBoundaryDiscretizationWidth(const Integer& ks, const Integer& kd, const Integer& boundaryDiscretizationWidth_)

This function is used to change the value of **boundaryDiscretizationWidth()** (§3.2.13).

3.2.15 const IntegerArray& boundaryCondition()

Dimensions: $(0{:}1, 0{:}2)$
boundaryCondition() holds the boundary condition flags, which indicate how each side of the grid is used; **boundaryCondition**(i,j) refers to the side of the grid corresponding to the coordinate value $r_j = i$ in the parameter space of the grid.

$$\textbf{boundaryCondition}(i,j) \quad \begin{cases} < 0 & \Rightarrow \text{ The domain is periodic in the coordinate } r_j. \\ = 0 & \Rightarrow \text{ The side corresponding to } r_j = i \text{ may only interpolate.} \\ > 0 & \Rightarrow \text{ The side corresponding to } r_j = i \text{ is part of the domain boundary.} \end{cases}$$

For the extra dimensions **numberOfDimensions**$() \le j \le 2$, if any, **boundaryCondition**$(i,j) = -1$. This data may be set using the function **setBoundaryCondition(ks,kd,boundaryCondition_)** (§3.2.16).

3.2.16 void setBoundaryCondition(const Integer& ks, const Integer& kd, const Integer& boundaryCondition_)

This function is used to change the value of **boundaryCondition()** (§3.2.15).

3.2.17 const IntegerArray& sharedBoundaryFlag() const

Dimensions: $(0{:}1, 0{:}2)$
sharedBoundaryFlags() holds the shared boundary flags, which may be used to indicate which sides of the grid correspond to the same feature of the domain boundary. Different features of the domain boundary may be distinguished from each other by their being separated by an edge or corner of the domain. Sides of grids which correspond to the same feature of the domain boundary ideally should match exactly where they overlap or should at least intersect only tangentially. In practice this is often impossible to ensure (especially in the case of grids whose mappings are defined discretely, for example, mappings based on splines), so this flag is useful in order to identify those cases where such was the intention. **sharedBoundaryFlag**(i,j) refers to the side of the grid corresponding to the coordinate value $r_j = i$ in the parameter space of the grid. A unique non-zero flag value should be assigned to **sharedBoundaryFlag**(i,j) for all of those sides of grids that correspond to the same feature of the domain boundary. For the extra dimensions **numberOfDimensions**$() \le j \le 2$, if any, **sharedBoundaryFlag**$(i, j) = 0$. This data may be set using the function **setSharedBoundaryFlag(ks,kd,sharedBoundaryFlag_)** (§3.2.18).

3.2.18 void setSharedBoundaryFlag(const Integer& ks, const Integer& kd, const Integer& sharedBoundaryFlag_)

This function is used to change the value of **sharedBoundaryFlag()** (§3.2.17).

3.2.19 const RealArray& sharedBoundaryTolerance() const

Dimensions: $(0{:}1, 0{:}2)$
sharedBoundaryTolerance() holds the shared boundary error tolerance, which indicates by how much the mapping that generates the grid may deviate from the ideal domain boundary on each side of the grid, normalized to the width of grid cells in the direction normal to the boundary. In the case where the ideal domain boundary is unknown, **sharedBoundaryTolerance** is a useful estimate of the mismatch between sides of the grid that correspond to the same feature of the domain boundary, as identified by **sharedBoundaryFlag()** (§3.2.17). **sharedBoundaryTolerance**(i,j) refers to the side of the grid corresponding to the coordinate value $r_j = i$ in the parameter space of the grid. This data may be set using the function **setSharedBoundaryTolerance(ks,kd,sharedBoundaryTolerance_)** (§3.2.20).

3.2.20 void setSharedBoundaryTolerance(const Integer& ks, const Integer& kd, const Real& sharedBoundaryTolerance_)

This function is used to change the value of **sharedBoundaryTolerance()** (§3.2.19).

3.2.21 const RealArray& gridSpacing() const

Dimensions: $(0{:}2)$
The grid spacing in the direction of the coordinate r_i in the parameter space of the grid is

$$\textbf{gridSpacing}(i) = \frac{1}{\textbf{gridIndexRange}(1,i) - \textbf{gridIndexRange}(0,i)}.$$

For the extra dimensions **numberOfDimensions**$() \le i \le 2$, if any, **gridSpacing**$(i) = 1$. This data is always recomputed by the function **update(what,how)** (§3.1.16).

3.2.22 const LogicalArray& isCellCentered() const

Dimensions: $(0{:}2)$
The flag **isCellCentered**(i) is **LogicalTrue** (non-zero) if and only if $i <$ **numberOfDimensions**$()$ and the grid is cell-centered in the direction corresponding to the coordinate r_i in the parameter space of the grid. Cell-centered in direction i means that discretization points lie at positions

$$r_i = \begin{cases} \big(j + \frac{1}{2} - \textbf{indexRange}(0,i)\big)\,\textbf{gridSpacing}(i) & \text{if } \textbf{isCellCentered}(i) \\ \big(j - \textbf{indexRange}(0,i)\big)\,\textbf{gridSpacing}(i) & \text{otherwise} \end{cases}$$

for **indexRange**$(0,i) \le j \le$ **indexRange**$(1,i)$. This data may be set using the function **setIsCellCentered(kd,isCellCentered_)** (§3.2.23).

3.2.23 void setIsCellCentered(const Integer& kd, const Logical& isCellCentered_)

This function is used to change the value of **isCellCentered()** (§3.2.22). If the new value is different from the old value, then all geometric data is destroyed. See also **changeToAllVertexCentered()** (§3.1.11) and **changeToAllCellCentered()** (§3.1.12).

3.2.24 const Logical& isAllCellCentered() const

This function returns a reference to a flag that is **LogicalTrue** (non-zero) if and only if the grid is cell-centered in all directions. This means that the discretization points are grid cell centers. This flag is always recomputed by the function **update(what,how)** (§3.1.16). See also **isCellCentered()** (§3.2.22).

3.2.25 const Logical& isAllVertexCentered() const

This function returns a reference to a flag that is **LogicalTrue** (non-zero) if and only if the grid is vertex-centered in all directions. This means that the discretization points are grid vertices. This flag is always recomputed by the function **update(what,how)** (§3.1.16). See also **isCellCentered()** (§3.2.22).

3.2.26 const IntegerArray& isPeriodic() const

Dimensions: $(0{:}2)$
isPeriodic(i) describes the periodicity of the grid, the mapping that generates the grid, and all **MappedGridFunctions** defined on the grid. **isPeriodic**(i) is zero (non-periodic) if and only if $i <$ **numberOfDimensions**$()$ and either the derivative of the mapping that generates the grid is not periodic in the direction corresponding to the coordinate r_i in the parameter space of the grid, or the periodicity of the domain does not correspond to the periodicity of the derivative of the mapping in this direction. Two cases exist when the periodicity of the mapping corresponds to the periodicity of the domain: either the mapping itself is periodic or it is not. (In the latter case the mapping differs from a periodic function by a linear function.) These two cases are distinguished by different non-zero values of the flag **isPeriodic**(i). All **MappedGridFunctions** defined on the grid should have the same periodicity as the mapping; in each direction where the mapping is periodic, all **MappedGridFunctions** should be periodic, and in each direction where the derivative of the mapping is periodic, the derivatives of all **MappedGridFunctions** should be periodic. The possible values of **isPeriodic**(i) are

$$\textbf{isPeriodic}(i) = \begin{cases} \textbf{Mapping::notPeriodic} = 0 & \text{The derivative of the mapping is not periodic.} \\ \textbf{Mapping::derivativePeriodic} & \text{The derivative is periodic but the mapping is not.} \\ \textbf{Mapping::functionPeriodic} & \text{The mapping is periodic.} \end{cases}$$

This data may be set using the function **setIsPeriodic(kd,isPeriodic_)** (§3.2.27).

3.2.27 void setIsPeriodic(const Integer& kd, const Mapping::periodicType& isPeriodic_)

This function is used to change the value of **isPeriodic()** (§3.2.26). If the new value is different from the old value, then all geometric data is destroyed. It is important to be sure that the setting of **isPeriodic** should be consistent with the topology of the mapping that defines the geometry of the grid.

3.2.28 const RealArray& minimumEdgeLength() const

Dimensions: $(0{:}2)$
minimumEdgeLength() holds the minimum grid cell-edge length over all cell edges in the interior and the boundary of the grid, for each coordinate direction in the parameter space of the grid. **minimumEdgeLength**(i) refers to edges of the cells corresponding to the coordinate direction r_i. This geometric data may be updated as in the following example. See also **THEminMaxEdgeLength** (§3.6.19) and **update(what,how)** (§3.1.16).

3.2.29 const RealArray& maximumEdgeLength() const

Dimensions: $(0{:}2)$
maximumEdgeLength() holds the maximum grid cell-edge length over all cell edges in the interior and the boundary of the grid, for each coordinate direction in the parameter space of the grid. **maximumEdgeLength**(i) refers to edges of the cells corresponding to the coordinate direction r_i. This geometric data may be updated as in the following example. See also **THEminMaxEdgeLength** (§3.6.19) and **update(what,how)** (§3.1.16).

3.2.30 const RealArray& boundingBox() const

Dimensions: $(0{:}\,1, 0{:}\,2)$
boundingBox() holds coordinate bounds for the grid vertices, including all boundary vertices but excluding any ghost vertices. This geometric data may be updated as in the following example. See also **THEboundingBox** (§3.6.20) and **update(what,how)** (§3.1.16).

3.2.31 const Integer* I1() const; const Integer* I2() const; const Integer* I3() const

I1(), **I2**() and **I3**() return pointers that may be used as arrays for indirect scalar addressing of **MappedGridFunction**s on periodic grids. This is useful for indices that lie in a discretization or interpolation stencil that extends outside the range of indices given by **indexRange()** (§3.2.3). For example, you might want to index **mask()** (§3.2.32) as $\textbf{mask}()(\textbf{I1}()[i], \textbf{I2}()[j], \textbf{I3}()[k])$ instead of $\textbf{mask}()(i,j,k)$ in case (i,j,k) might lie outside the range of discretization points of the grid. The indirect addressing arrays are defined for indices which lie outside by at most sixteen. If you anticipate using stencils wider than this limit would accomodate, then we can increase it for you.

3.2.32 IntegerMappedGridFunction& mask()

Dimensions: $(d_{00}{:}\,d_{10}, d_{01}{:}\,d_{11}, d_{02}{:}\,d_{12})$, where $d_{ij} = \textbf{dimension}(i,j)$.
mask() holds a flag for each point (*e.g.,* vertex or cell-center), which indicates how that point should be used. Various bits of the mask may be tested by ANDing the mask with any of the constants **ISdiscretizationPoint** (§3.6.27), **ISinterpolationPoint** (§3.6.28), **ISghostPoint** (§3.6.29), **ISinteriorBoundaryPoint** (§3.6.30), **USESbackupRules** (§3.6.31), **IShiddenByRefinement** (§3.6.32) or **ISusedPoint** (§3.6.37), and checking if the result is non-zero.

$$\textbf{mask}()(i,j,k) \begin{cases} \& \ \textbf{ISdiscretizationPoint} \neq 0 & \Rightarrow \text{ discretization point} \\ \& \ \textbf{ISinterpolationPoint} \neq 0 & \Rightarrow \text{ interpolation point} \\ \& \ \textbf{ISghostPoint} \neq 0 & \Rightarrow \text{ ghost point near a disc. or interp. boundary point.} \\ \& \ \textbf{ISinteriorBoundaryPoint} \neq 0 & \Rightarrow \text{ boundary point in the interior of another grid} \\ \& \ \textbf{USESbackupRules} \neq 0 & \Rightarrow \text{ discretization or interpolation uses backup rules} \\ \& \ \textbf{IShiddenByRefinement} \neq 0 & \Rightarrow \text{ hidden by points on a refinement grid} \\ \& \ \textbf{ISusedPoint} = 0 & \Rightarrow \text{ neither discretization nor interpolation point} \end{cases}$$

This data may be updated as in the following example. See also **THEmask** (§3.6.1) and **update(what,how)** (§3.1.16).

3.2.33 RealMappedGridFunction& vertex()

Dimensions: $(d_{00}{:}\,d_{10}, d_{01}{:}\,d_{11}, d_{02}{:}\,d_{12}, 0{:}\,n_1)$, where $d_{ij} = \textbf{dimension}(i,j)$ and $n_1 = \textbf{numberOfDimensions}() - 1$.
vertex() holds the coordinates of the vertices of the grid, including any ghost vertices (which lie outside the grid).

$$\textbf{vertex}()(i_0,i_1,i_2,*) = \boldsymbol{g}(\mathbf{r}), \quad \text{where} \quad r_j = \big(i_j - \textbf{indexRange}(0,j)\big)\,\textbf{gridSpacing}(j),$$

$\textbf{dimension}(0,j) \le i_j \le \textbf{dimension}(1,j)$, and $\boldsymbol{g}$ is the mapping that generates the grid. This geometric data may be updated as in the following example. See also **THEvertex** (§3.6.2) and **update(what,how)** (§3.1.16).

3.2.34 RealMappedGridFunction& center()

Dimensions: $(d_{00}{:}\,d_{10}, d_{01}{:}\,d_{11}, d_{02}{:}\,d_{12}, 0{:}\,n_1)$, where $d_{ij} = \textbf{dimension}(i,j)$ and $n_1 = \textbf{numberOfDimensions}() - 1$.
center() holds the coordinates of the discretization points of the grid, (*e.g.,* the vertices or the grid cell-centers), including any ghost points (which lie outside the grid).

$$\textbf{center}()(i_0,i_1,i_2,*) = \boldsymbol{g}(\mathbf{r}), \quad \text{where} \quad r_j = \begin{cases} \big(i_j + \frac{1}{2} - \textbf{indexRange}(0,j)\big)\,\textbf{gridSpacing}(j) & \text{if } \textbf{isCellCentered}(j) \\ \big(i_j - \textbf{indexRange}(0,j)\big)\,\textbf{gridSpacing}(j) & \text{otherwise,} \end{cases}$$

$\textbf{dimension}(0,j) \le i_j \le \textbf{dimension}(1,j)$, and $\boldsymbol{g}$ is the mapping that generates the grid. If **center**() is updated using a discrete approximation, then it is not computed using the mapping, but is instead computed by averaging **corner()** (§3.2.35) in each direction:

$$\textbf{center}()(i_0,i_1,i_2,*) = \Big(\prod_j \mu_{+j}\Big)\textbf{corner}()(i_0,i_1,i_2,*).$$

Warning: In the case of a vertex-centered grid, ***center()*** *is not computed by averaging* ***corner****, but is equal to, and may be aliased to,* **vertex()** (§3.2.33). This geometric data may be updated as in the following example. See also **THEcenter** (§3.6.3) and **update(what,how)** (§3.1.16).

3.2.35 RealMappedGridFunction& corner()

Dimensions: $(d_{00}{:}\, d_{10}, d_{01}{:}\, d_{11}, d_{02}{:}\, d_{12}, 0{:}\, n_1)$, where $d_{ij} =$ **dimension**(i,j) and $n_1 =$ **numberOfDimensions**$() - 1$. **corner**() holds the coordinates of the corners of cells centered at the discretization points of the grid, (*e.g.*, the vertices or the grid cell-centers), including any ghost points (which lie outside the grid).

$$\textbf{corner}()(i_0,i_1,i_2,*) = \boldsymbol{g}(\mathbf{r}), \quad \text{where} \quad r_j = \begin{cases} \big(i_j - \textbf{indexRange}(0,j)\big)\,\textbf{gridSpacing}(j) & \text{if } \textbf{isCellCentered}(j) \\ \big(i_j - \frac{1}{2} - \textbf{indexRange}(0,j)\big)\,\textbf{gridSpacing}(j) & \text{otherwise,} \end{cases}$$

dimension$(0,j) \le i_j \le$ **dimension**$(1,j)$, and $\boldsymbol{g}$ is the mapping that generates the grid. *Note: In the case of a cell-centered grid,* ***corner()*** *is equal to, and may be aliased to,* **vertex()** (§3.2.33). This geometric data may be updated as in the following example. See also **THEcorner** (§3.6.4) and **update(what,how)** (§3.1.16).

3.2.36 RealMappedGridFunction& vertexDerivative()

Dimensions: $(d_{00}{:}\, d_{10}, d_{01}{:}\, d_{11}, d_{02}{:}\, d_{12}, 0{:}\, n_1, 0{:}\, n_1)$, where $d_{ij} =$ **dimension**(i,j) and $n_1 =$ **numberOfDimensions**$() - 1$. **vertexDerivative**() holds the derivative of the mapping at the vertices of the grid, including any ghost vertices (which lie outside the grid).

$$\textbf{vertexDerivative}()(i_0,i_1,i_2,*,j) = \frac{\partial \boldsymbol{g}}{\partial r_j}, \quad \text{where} \quad r_j = \big(i_j - \textbf{indexRange}(0,j)\big)\,\textbf{gridSpacing}(j),$$

dimension$(0,j) \le i_j \le$ **dimension**$(1,j)$, and $\boldsymbol{g}$ is the mapping that generates the grid. If **vertexDerivative**() is updated using a discrete approximation, then it is not computed using the mapping, but is instead computed by centered finite differences of **vertex**:

$$\textbf{vertexDerivative}()(i_0,i_1,i_2,*,j) = \frac{1}{2\Delta r_j}\Delta_{0j}\textbf{vertex}()(i_0,i_1,i_2,*),$$

where $\Delta r_j =$ **gridSpacing**(j). This geometric data may be updated as in the following example. See also **THEvertexDerivative** (§3.6.5) and **update(what,how)** (§3.1.16).

3.2.37 RealMappedGridFunction& centerDerivative()

Dimensions: $(d_{00}{:}\, d_{10}, d_{01}{:}\, d_{11}, d_{02}{:}\, d_{12}, 0{:}\, n_1, 0{:}\, n_1)$, where $d_{ij} =$ **dimension**(i,j) and $n_1 =$ **numberOfDimensions**$() - 1$. **centerDerivative**() holds the derivative of the mapping at the discretization points of the grid, including any ghost points (which lie outside the grid).

$$\textbf{centerDerivative}()(i_0,i_1,i_2,*,j) = \frac{\partial \boldsymbol{g}}{\partial r_j}, \quad \text{where}$$

$$r_j = \begin{cases} \big(i_j + \frac{1}{2} - \textbf{indexRange}(0,j)\big)\,\textbf{gridSpacing}(j) & \text{if } \textbf{isCellCentered}(j) \\ \big(i_j - \textbf{indexRange}(0,j)\big)\,\textbf{gridSpacing}(j) & \text{otherwise,} \end{cases}$$

dimension$(0,j) \le i_j \le$ **dimension**$(1,j)$, and $\boldsymbol{g}$ is the mapping that generates the grid. If **centerDerivative**() is updated using a discrete approximation, then it is not computed using the mapping, but is instead **corner()** (§3.2.35):

$$\textbf{centerDerivative}()(i_0,i_1,i_2,*,j) = \Big(\prod_{k\ne j}\mu_{+k}\Big)\frac{1}{\Delta r_j}\Delta_{+j}\textbf{corner}()(i_0,i_1,i_2,*).$$

where $\Delta r_j =$ **gridSpacing**(j). *Warning: In the case of a vertex-centered grid,* ***centerDerivative()*** *is not computed by differencing and averaging* ***corner()****, but is equal to, and may be aliased to,* **vertexDerivative()** (§3.2.36). This geometric data may be updated as in the following example. See also **THEcenterDerivative** (§3.6.6) and **update(what,how)** (§3.1.16).

3.2.38 RealMappedGridFunction& inverseVertexDerivative()

Dimensions: $(d_{00}{:}\, d_{10}, d_{01}{:}\, d_{11}, d_{02}{:}\, d_{12}, 0{:}\, n_1, 0{:}\, n_1)$, where $d_{ij} =$ **dimension**(i,j) and $n_1 =$ **numberOfDimensions**$() - 1$. **inverseVertexDerivative**() holds the inverse of the derivative of the mapping at the vertices of the grid, including any ghost vertices (which lie outside the grid).

$$[\textbf{inverseVertexDerivative}()(i_0,i_1,i_2,*,*)] = [\textbf{vertexDerivative}()(i_0,i_1,i_2,*,*)]^{-1} \quad = \left[\frac{\partial \mathbf{r}}{\partial \mathbf{x}}\right].$$

This geometric data may be updated as in the following example. See also **vertexDerivative()** (§3.2.36), **THEinverseVertexDerivative** (§3.6.7) and **update(what,how)** (§3.1.16).

3.2.39 RealMappedGridFunction& inverseCenterDerivative()

Dimensions: $(d_{00}{:}\,d_{10}, d_{01}{:}\,d_{11}, d_{02}{:}\,d_{12}, 0{:}\,n_1, 0{:}\,n_1)$, where $d_{ij} =$ **dimension**(i,j) and $n_1 =$ **numberOfDimensions**$() - 1$.
inverseCenterDerivative() holds the inverse of the derivative of the mapping at the discretization points of the grid, (*e.g.,* the vertices or the grid cell-centers), including any ghost points (which lie outside the grid).

$$[\textbf{inverseCenterDerivative}()(i_0,i_1,i_2,*,*)] = [\textbf{centerDerivative}()(i_0,i_1,i_2,*,*)]^{-1} \quad = \left[\frac{\partial \mathbf{r}}{\partial \mathbf{x}}\right].$$

This geometric data may be updated as in the following example. See also **centerDerivative()** (§3.2.37), **THEinverseCenterDerivative** (§3.6.8) and **update(what,how)** (§3.1.16).

3.2.40 RealMappedGridFunction& vertexJacobian()

Dimensions: $(d_{00}{:}\,d_{10}, d_{01}{:}\,d_{11}, d_{02}{:}\,d_{12})$, where $d_{ij} =$ **dimension**(i,j).
vertexJacobian() holds the determinant of the derivative of the mapping at the vertices of the grid, including any ghost vertices (which lie outside the grid).

$$\textbf{vertexJacobian}()(i_0,i_1,i_2) = \det\left[\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{r}}\right], \quad \text{where} \quad r_j = \big(i_j - \textbf{indexRange}(0,j)\big)\,\textbf{gridSpacing}(j),$$

dimension$(0,j) \le i_j \le$ **dimension**$(1,j)$, and $\boldsymbol{g}$ is the mapping that generates the grid. If **vertexJacobian**() is updated using a discrete approximation, then it is not computed using the mapping, but is instead computed using the same approximation to the derivative as that used for **vertexJacobian()** (§3.2.40). This geometric data may be updated as in the following example. See also **vertexDerivative()** (§3.2.36), **THEvertexJacobian** (§3.6.9) and **update(what,how)** (§3.1.16).

3.2.41 RealMappedGridFunction& centerJacobian()

Dimensions: $(d_{00}{:}\,d_{10}, d_{01}{:}\,d_{11}, d_{02}{:}\,d_{12})$, where $d_{ij} =$ **dimension**(i,j).
centerJacobian() holds the determinant of the derivative of the mapping at the discretization points of the grid, including any ghost points (which lie outside the grid).

$$\textbf{centerJacobian}()(i_0,i_1,i_2) = \det\left[\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{r}}\right], \quad \text{where}$$

$$r_j = \begin{cases} \big(i_j + \frac{1}{2} - \textbf{indexRange}(0,j)\big)\,\textbf{gridSpacing}(j) & \text{if } \textbf{isCellCentered}(j) \\ \big(i_j - \textbf{indexRange}(0,j)\big)\,\textbf{gridSpacing}(j) & \text{otherwise,} \end{cases}$$

dimension$(0,j) \le i_j \le$ **dimension**$(1,j)$, and $\boldsymbol{g}$ is the mapping that generates the grid. If **centerJacobian**() is updated using a discrete approximation, then it is not computed using the mapping, but is instead computed using the same approximation to the derivative as that used for **centerDerivative()** (§3.2.37). This geometric data may be updated as in the following example. See also **centerDerivative()** (§3.2.37), **THEcenterJacobian** (§3.6.10) and **update(what,how)** (§3.1.16).

3.2.42 RealMappedGridFunction& cellVolume()

Dimensions: $(d_{00}{:}\,d_{10}, d_{01}{:}\,d_{11}, d_{02}{:}\,d_{12})$, where $d_{ij} =$ **dimension**(i,j).
cellVolume() holds the volumes of cells centered at the discretization points of the grid, including any ghost cells (which lie outside the grid).

$$\textbf{cellVolume}()(i_0,i_1,i_2) = \Big(\prod_j \Delta r_j\Big) \det\left[\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{r}}\right], \quad \text{where}$$

$$r_j = \begin{cases} \big(i_j + \frac{1}{2} - \textbf{indexRange}(0,j)\big)\,\textbf{gridSpacing}(j) & \text{if } \textbf{isCellCentered}(j) \\ \big(i_j - \textbf{indexRange}(0,j)\big)\,\textbf{gridSpacing}(j) & \text{otherwise,} \end{cases}$$

$\Delta r_j =$ **gridSpacing**(j), **dimension**$(0,j) \le i_j \le$ **dimension**$(1,j)$, and $\boldsymbol{g}$ is the mapping that generates the grid. **cellVolume**() has the same sign as the determinant of the derivative of the mapping. (*I.e.,* it may be negative.) If **cellVolume**() is updated using a discrete approximation, then it is not computed using the mapping, but is instead computed in one dimension as the distance between surrounding corners, approximated in two dimensions by the area of the polygon bounded by the surrounding corners, and approximated in three dimensions by the volume of the solid bounded by the surrounding corners, with the approximation that the four corners of each face are assumed to be coplanar. This geometric data may be updated as in the following example. See also **THEcellVolume** (§3.6.11) and **update(what,how)** (§3.1.16).

3.2.43 RealMappedGridFunction& faceNormal()

Dimensions: $(d_{00}{:}\, d_{10}, d_{01}{:}\, d_{11}, d_{02}{:}\, d_{12}, 0{:}\, n_1, 0{:}\, n_1)$, where $d_{ij} =$ **dimension**(i,j) and $n_1 =$ **numberOfDimensions**$() - 1$. **faceNormal()** holds vectors normal to faces of cells centered at the discretization points (including any ghost cell faces), normalized to the cell-face area. The normal to cell face (i_0, i_1, i_2) corresponding to constant r_j is given by

$$\textbf{faceNormal}()(i_0,i_1,i_2,*,j) = \Delta r_k \Delta r_l \frac{\partial \mathbf{g}}{\partial r_k} \times \frac{\partial \mathbf{g}}{\partial r_l}, \quad \text{where } k=(j+1) \bmod 3,\ l=(j+2) \bmod 3,$$

where $\Delta r_j = 1$ for $j > n_0$, $\frac{\partial g_i}{\partial r_j} = \delta_{ij}$ for $i > n_0$ or $j > n_0$, $n_0 =$ **numberOfDimensions**$() - 1$, $\Delta r_j =$ **gridSpacing**(j),

$$r_k = \begin{cases} \big(i_k - \textbf{indexRange}(0,k)\big)\,\textbf{gridSpacing}(k) & \text{if } k=j \text{ and } \textbf{isCellCentered(k)}, \text{ or else} \\ \big(i_k + \frac{1}{2} - \textbf{indexRange}(0,k)\big)\,\textbf{gridSpacing}(k) & \text{if } k=j, \text{ or} \\ \big(i_k - \frac{1}{2} - \textbf{indexRange}(0,k)\big)\,\textbf{gridSpacing}(k) & \text{if } \textbf{isCellCentered(k)}, \text{ or else} \\ \big(i_k - \textbf{indexRange}(0,k)\big)\,\textbf{gridSpacing}(k) & \text{otherwise,} \end{cases}$$

dimension$(0,k) \le i_k \le$ **dimension**$(1,k)$, and $\boldsymbol{g}$ is the mapping that generates the grid. In particular, **faceNormal**$()(i_0,i_1,i_2,0,0) = 1$ if **numberOfDimensions**$() = 1$. If **faceNormal()** is updated using a discrete approximation, then the derivatives are not computed using the mapping, but are instead computed by centered finite differences of **corner()** (§3.2.35), (averaged in three dimensions to the centers of cell faces). As a result, in two dimensions,

$$\begin{aligned} \textbf{faceNormal}()(i,j,k,*,0) &= \Delta_{+j}\textbf{corner}(i,j,k,*) \\ \textbf{faceNormal}()(i,j,k,*,1) &= -\Delta_{+i}\textbf{corner}(i,j,k,*) \end{aligned}$$

and in three dimensions,

$$\begin{aligned} \textbf{faceNormal}()(i_0,i_1,i_2,*,j) &= \mu_{+i_l}\Delta_{+i_k}\textbf{corner}(i_0,i_i,i_2,*) \times \mu_{+i_k}\Delta_{+i_l}\textbf{corner}(i_0,i_i,i_2,*) \\ &= \frac{1}{2}\Delta_{\nearrow +i_k+i_l}\textbf{corner}(i_0,i_1,i_2,*) \times \Delta_{\nwarrow +i_l+i_k}\textbf{corner}(i_0,i_1,i_2,*) \end{aligned}$$

where $k=(j+1) \bmod 3$, $l=(j+2) \bmod 3$, $\Delta_{\nearrow +i+j}u_{ij} \equiv u_{i+1,j+1} - u_{ij}$ and $\Delta_{\nwarrow +i+j}u_{ij} \equiv u_{i,j+1} - u_{i+1,j}$. This geometric data may be updated as in the following example. See also **THEfaceNormal** (§3.6.12) and **update(what,how)** (§3.1.16).

3.2.44 RealMappedGridFunction& centerNormal()

Dimensions: $(d_{00}{:}\, d_{10}, d_{01}{:}\, d_{11}, d_{02}{:}\, d_{12}, 0{:}\, n_1, 0{:}\, n_1)$, where $d_{ij} =$ **dimension**(i,j) and $n_1 =$ **numberOfDimensions**$() - 1$. The normal to the surface corresponding to constant r_j and passing through the discretization point (i_0, i_1, i_2), normalized to the area of that portion of this surface which corresponds one cell, is given by

$$\textbf{centerNormal}()(i_0,i_1,i_2,*,j) = \Delta r_k \Delta r_l \frac{\partial \mathbf{g}}{\partial r_k} \times \frac{\partial \mathbf{g}}{\partial r_l}, \quad \text{where } k=(j+1) \bmod 3,\ l=(j+2) \bmod 3,$$

where $\Delta r_j = 1$ for $j > n_0$, $\frac{\partial g_i}{\partial r_j} = \delta_{ij}$ for $i > n_0$ or $j > n_0$, $n_0 =$ **numberOfDimensions**$() - 1$, $\Delta r_j =$ **gridSpacing**(j),

$$r_k = \begin{cases} \big(i_k + \frac{1}{2} - \textbf{indexRange}(0,k)\big)\,\textbf{gridSpacing}(k) & \text{if } \textbf{isCellCentered}(k) \\ \big(i_k - \textbf{indexRange}(0,k)\big)\,\textbf{gridSpacing}(k) & \text{otherwise,} \end{cases}$$

dimension$(0,k) \le i_k \le$ **dimension**$(1,k)$, and $\boldsymbol{g}$ is the mapping that generates the grid. In particular, **centerNormal**$()(i_0,i_1,i_2,0,0) = 1$ if **numberOfDimensions**$() = 1$. In fact, **centerNormal()** is related to **inverseCenterDerivative()** (§3.2.39) and **centerJacobian()** (§3.2.41):

$$[\textbf{centerNormal}()(i_0,i_1,i_2,*,*)] = \Big(\prod_j \Delta r_j\Big)\textbf{centerJacobian}()(i_0,i_1,i_2)\,[\textbf{inverseCenterDerivative}()(i_0,i_1,i_2,*,*)]^{\mathrm{T}}.$$

If **centerNormal()** is updated using a discrete approximation, then it is obtained by averaging **faceNormal()** (§3.2.43) from the face-centers to the discretization points. This geometric data may be updated as in the following example. See also **THEcenterNormal** (§3.6.13) and **update(what,how)** (§3.1.16).

3.2.45 RealMappedGridFunction& faceArea()

Dimensions: $(d_{00}{:}\, d_{10}, d_{01}{:}\, d_{11}, d_{02}{:}\, d_{12}, 0{:}\, n_1)$, where $d_{ij} =$ **dimension**(i,j) and $n_1 =$ **numberOfDimensions**$() - 1$.
faceArea() holds the areas of faces of cells centered at the discretization points (including any ghost cell faces). The area of the cell face (i_0, i_1, i_2) corresponding to constant r_j is given by

$$\textbf{faceArea}()(i_0,i_1,i_2,j) = \big|\textbf{faceNormal}()(i_0,i_1,i_2,*,j)\big|.$$

This geometric data may be updated as in the following example. See also **faceNormal()** (§3.2.43), **THEfaceArea** (§3.6.14) and **update(what,how)** (§3.1.16).

3.2.46 RealMappedGridFunction& centerArea()

Dimensions: $(d_{00}{:}\, d_{10}, d_{01}{:}\, d_{11}, d_{02}{:}\, d_{12}, 0{:}\, n_1)$, where $d_{ij} =$ **dimension**(i,j) and $n_1 =$ **numberOfDimensions**$() - 1$.
centerArea() holds the areas of those portions of surfaces corresponding to a constant parameter value, passing through the discretization points, which are contained within the cells centered at these points (including any ghost cells). The area of the portion of the surface passing through the cell centered at the discretization point (i_0, i_1, i_2) and corresponding to constant r_j is given by

$$\textbf{centerArea}()(i_0,i_1,i_2,j) = \big|\textbf{centerNormal}()(i_0,i_1,i_2,*,j)\big|.$$

This geometric data may be updated as in the following example. See also **centerNormal()** (§3.2.44), **THEcenterArea** (§3.6.15) and **update(what,how)** (§3.1.16).

3.2.47 RealMappedGridFunction& vertexBoundaryNormal(const Integer& k, const Integer& l)

Dimensions of vertexBoundaryNormal(*k*,*l*): $(d_{00}{:}\, d_{10}, d_{01}{:}\, d_{11}, d_{02}{:}\, d_{12}, 0{:}\, n_1)$, where

$$d_{ij} = \begin{cases} \textbf{dimension}(k,j) & \text{if } j = l \\ \textbf{dimension}(i,j) & \text{otherwise} \end{cases}$$

and $n_1 =$ **numberOfDimensions**$() - 1$.
vertexBoundaryNormal holds unit outward normal vectors to the boundary at the boundary vertices. The normal corresponding to the side of the grid where $r_j = i$ is given by

$$\textbf{vertexBoundaryNormal}(i,j)(i_0,i_1,i_2,*) = \pm(-1)^{i+1}\frac{\frac{\partial \mathbf{g}}{\partial r_k}\times\frac{\partial \mathbf{g}}{\partial r_l}}{\left|\frac{\partial \mathbf{g}}{\partial r_k}\times\frac{\partial \mathbf{g}}{\partial r_l}\right|}, \quad \text{where } k=(j+1) \bmod 3,\ l=(j+2) \bmod 3,$$

$\frac{\partial g_i}{\partial r_j} = \delta_{ij}$ for $i > n_0$ or $j > n_0$, $n_0 =$ **numberOfDimensions**$() - 1$,

$$r_m = \big(i_m - \textbf{indexRange}(0,m)\big)\,\textbf{gridSpacing}(m),$$

dimension$(0,m) \le i_m \le$ **dimension**$(1,m)$, and $\boldsymbol{g}$ is the mapping that generates the grid. In particular, **vertexBoundaryNormal**$(i,0)(i_0,i_1,i_2,0) = \pm(-1)^{i+1}$ if **numberOfDimensions**$() = 1$. The upper sign is taken if the coordinate system is right-handed and the lower sign if it is left-handed; the sign taken is that of the jacobian of the mapping, $\det\left[\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{r}}\right]$ at the center of the grid $r_0 = r_1 = r_2 = \frac{1}{2}$. If **vertexBoundaryNormal** is updated using a discrete approximation, then the derivatives are not computed using the mapping, but are instead computed by centered finite differences of **vertex**. As a result, in two dimensions,

$$\begin{aligned}
\textbf{vertexBoundaryNormal}(i,0)(i_0,i_1,i_2,*) &= \pm(-1)^{i+1}\frac{\Delta_{0i_1}\textbf{vertex}()(i_0,i_1,i_2,*)}{\left|\Delta_{0i_1}\textbf{vertex}()(i_0,i_1,i_2,*)\right|} \\
\textbf{vertexBoundaryNormal}(i,1)(i_0,i_1,i_2,*) &= \mp(-1)^{i+1}\frac{\Delta_{0i_0}\textbf{vertex}()(i_0,i_1,i_2,*)}{\left|\Delta_{0i_0}\textbf{vertex}()(i_0,i_1,i_2,*)\right|},
\end{aligned}$$

and in three dimensions,

$$\textbf{vertexBoundaryNormal}(i,j)(i_0,i_1,i_2,*) = \pm(-1)^{i+1}\frac{\Delta_{0i_k}\textbf{vertex}(i_0,i_i,i_2,*)\times\Delta_{0i_l}\textbf{vertex}(i_0,i_i,i_2,*)}{\left|\Delta_{0i_k}\textbf{vertex}(i_0,i_i,i_2,*)\times\Delta_{0i_l}\textbf{vertex}(i_0,i_i,i_2,*)\right|}$$

where $k = (j+1) \bmod 3$ and $l = (j+2) \bmod 3$. This geometric data may be updated as in the following example. See also **THEvertexBoundaryNormal** (§3.6.16) and **update(what,how)** (§3.1.16).

3.2.48 RealMappedGridFunction& centerBoundaryNormal(const Integer& k, const Integer& l)

Dimensions of centerBoundaryNormal(*k*,*l*): $(d_{00}\colon d_{10}, d_{01}\colon d_{11}, d_{02}\colon d_{12}, 0\colon n_1)$, where

$$d_{ij} = \begin{cases} \textbf{dimension}(k,j) & \text{if } j = l \\ \textbf{dimension}(i,j) & \text{otherwise} \end{cases}$$

and $n_1 =$ **numberOfDimensions**$() - 1$.
centerBoundaryNormal holds unit outward normal vectors to the boundary at the discretization points of the boundary. Note that for a cell-centered grid, these points are not the cell-centers of boundary cells, but are the centers of the faces of boundary cells. The normal corresponding to the side of the grid where $r_j = i$ is given by

$$\textbf{centerBoundaryNormal}(i,j)(i_0,i_1,i_2,*) = \pm(-1)^{i+1} \frac{\frac{\partial \mathbf{g}}{\partial r_k} \times \frac{\partial \mathbf{g}}{\partial r_l}}{\left| \frac{\partial \mathbf{g}}{\partial r_k} \times \frac{\partial \mathbf{g}}{\partial r_l} \right|}, \quad \text{where } k = (j+1) \bmod 3,\ l = (j+2) \bmod 3,$$

$\frac{\partial g_i}{\partial r_j} = \delta_{ij}$ for $i > n_0$ or $j > n_0$, $n_0 =$ **numberOfDimensions**$() - 1$,

$$r_m = \begin{cases} i & \text{if } m = j, \text{ or else} \\ \left(i_m + \frac{1}{2} - \textbf{indexRange}(0,m)\right) \textbf{gridSpacing}(m) & \text{if } \textbf{isCellCentered}(m) \\ \left(i_m - \textbf{indexRange}(0,m)\right) \textbf{gridSpacing}(m) & \text{otherwise,} \end{cases}$$

dimension$(0,m) \le i_m \le$ **dimension**$(1,m)$, and $\boldsymbol{g}$ is the mapping that generates the grid. In particular, **centerBoundaryNormal**$(i,0)(i_0,i_1,i_2,0) = \pm(-1)^{i+1}$ if **numberOfDimensions**$() = 1$. The upper sign is taken if the coordinate system is right-handed and the lower sign if it is left-handed; the sign taken is that of the jacobian of the mapping, $\det\left[\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{r}}\right]$ at the center of the grid $r_0 = r_1 = r_2 = \frac{1}{2}$. If **centerBoundaryNormal** is updated using a discrete approximation, then the derivatives are not computed using the mapping, but are instead computed by centered finite differences of **corner**, for the case of a cell-centered grid. As a result, in two dimensions,

$$\begin{aligned} \textbf{centerBoundaryNormal}(i,0)(i_0,i_1,i_2,*) &= \pm(-1)^{i+1} \frac{\Delta_{+i_1}\textbf{corner}()(i_0,i_1,i_2,*)}{\left|\Delta_{+i_1}\textbf{corner}()(i_0,i_1,i_2,*)\right|} \\ \textbf{centerBoundaryNormal}(i,1)(i_0,i_1,i_2,*) &= \mp(-1)^{i+1} \frac{\Delta_{+i_0}\textbf{corner}()(i_0,i_1,i_2,*)}{\left|\Delta_{+i_0}\textbf{corner}()(i_0,i_1,i_2,*)\right|}. \end{aligned}$$

In three dimensions,

$$\begin{aligned} &\textbf{centerBoundaryNormal}(i,j)(i_0,i_1,i_2,*) \\ &\quad = \pm(-1)^{i+1} \frac{\mu_{+i_l}\Delta_{+i_k}\textbf{corner}(i_0,i_i,i_2,*) \times \mu_{+i_k}\Delta_{+i_l}\textbf{corner}(i_0,i_i,i_2,*)}{\left|\mu_{+i_l}\Delta_{+i_k}\textbf{corner}(i_0,i_i,i_2,*) \times \mu_{+i_k}\Delta_{+i_l}\textbf{corner}(i_0,i_i,i_2,*)\right|} \\ &\quad = \pm(-1)^{i+1} \frac{\Delta_{\nearrow +i_k+i_l}\textbf{corner}(i_0,i_1,i_2,*) \times \Delta_{\nwarrow +i_l+i_k}\textbf{corner}(i_0,i_1,i_2,*)}{\left|\Delta_{\nearrow +i_k+i_l}\textbf{corner}(i_0,i_1,i_2,*) \times \Delta_{\nwarrow +i_l+i_k}\textbf{corner}(i_0,i_1,i_2,*)\right|}, \end{aligned}$$

where $k = (j+1) \bmod 3$, $l = (j+2) \bmod 3$, $\Delta_{\nearrow +i+j}u_{ij} \equiv u_{i+1,j+1} - u_{ij}$ and $\Delta_{\nwarrow +i+j}u_{ij} \equiv u_{i,j+1} - u_{i+1,j}$. If the grid is not cell-centered in the normal direction j, then the difference approximations are applied not to **corner** itself, but are applied instead to what **corner** would be, were the grid cell-centered in the the normal direction. *Warning: In the case of a vertex-centered grid,* ***centerBoundaryNormal*** *is not computed by differencing* ***corner****, but is equal to, and may be aliased to,* **vertexBoundaryNormal** (§3.2.47). This geometric data may be updated as in the following example. See also **THEcenterBoundaryNormal** (§3.6.17) and **update(what,how)** (§3.1.16).

3.2.49 RealMappedGridFunction& centerBoundaryTangent(const Integer& k, const Integer& l)

Dimensions of centerBoundaryTangent(*k*,*l*): $(d_{00}\colon d_{10}, d_{01}\colon d_{11}, d_{02}\colon d_{12}, 0\colon n_1, 0\colon n_2)$, where

$$d_{ij} = \begin{cases} \textbf{dimension}(k,j) & \text{if } j = l \\ \textbf{dimension}(i,j) & \text{otherwise,} \end{cases}$$

$n_1 =$ **numberOfDimensions**$() - 1$ and $n_2 =$ **numberOfDimensions**$() - 2$.
centerBoundaryTangent holds unit tangent vectors to the boundary surface at the discretization points of the boundary. Note

that for a cell-centered grid, these points are not the cell-centers of boundary cells, but are the centers of the faces of boundary cells. The tangents corresponding to the side of the grid where $r_j = i$ are given by

$$\textbf{centerBoundaryTangent}(l,i)(i_0,i_1,i_2,*,j) = \frac{\frac{\partial \mathbf{g}}{\partial r_k}}{\left|\frac{\partial \mathbf{g}}{\partial r_k}\right|}, \quad \text{where } k = (i+j+1) \bmod \textbf{numberOfDimensions}(),$$

$$r_m = \begin{cases} l & \text{if } m = i, \text{ or else} \\ \left(i_m + \frac{1}{2} - \textbf{indexRange}(0,m)\right) \textbf{gridSpacing}(m) & \text{if } \textbf{isCellCentered}(m) \\ \left(i_m - \textbf{indexRange}(0,m)\right) \textbf{gridSpacing}(m) & \text{otherwise,} \end{cases}$$

dimension$(0,m) \le i_m \le$ **dimension**$(1,m)$, and $\boldsymbol{g}$ is the mapping that generates the grid. If **centerBoundaryTangent** is updated using a discrete approximation, then the derivatives are not computed using the mapping, but are instead computed by centered finite differences of **corner**, for the case of a cell-centered grid. As a result, in two dimensions,

$$\textbf{centerBoundaryTangent}(*,i)(i_0,i_1,i_2,*,j) = \frac{\Delta_{+i_k}\textbf{corner}(i_0,i_1,i_2,*)}{\left|\Delta_{+i_k}\textbf{corner}(i_0,i_1,i_2,*)\right|},$$

and in three dimensions,

$$\textbf{centerBoundaryTangent}(*,i)(i_0,i_1,i_2,*,j) = \frac{\mu_{+i_l}\Delta_{+k}\textbf{corner}(i_0,i_i,i_2,*)}{\left|\mu_{+i_l}\Delta_{+k}\textbf{corner}(i_0,i_i,i_2,*)\right|},$$

where $k = (i+j+1) \bmod 3$ and $l = (k+1) \bmod 3$. If the grid is not cell-centered in the normal direction j, then the difference approximations are applied not to **corner** itself, but are applied instead to what **corner** would be, were the grid cell-centered in the the normal direction. This geometric data may be updated as in the following example. See also **THEcenterBoundaryTangent** (§3.6.18) and **update(what,how)** (§3.1.16).

3.2.50 MappingRC& mapping()

mapping() is the reference-counted mapping that generates the grid. The mapping may be replaced as in the following example. See also **reference(const Mapping& x)** (§3.1.8) and **reference(const MappingRC& x)** (§3.1.9).

3.2.51 const Box& box() const

Class **Box** comes from Boxlib, and is used for adaptive mesh refinement. This function returns a reference to a **Box** that describes this **MappedGrid**.

3.2.52 virtual aString getClassName() const

Get the class name of the most-derived class for this object.

3.2.53 MappedGridData* operator->()

Access the reference-counted data.

3.2.54 MappedGridData& operator*()

Access the pointer to the reference-counted data.

3.3 Public member functions called only from derived classes

It is recommended that these functions be called only from derived classes.

3.3.1 void reference(MappedGridData& x)

Make a reference to an object of type **MappedGridData**. This **MappedGrid** uses **x** for its data. It is recommended that this function be called only from derived classes.

3.3.2 void updateReferences(const Integer what = EVERYTHING)

Update references to the reference-counted data. It is recommended that this function be called only from derived classes.

3.3.3 void setNumberOfDimensions(const Integer& numberOfDimensions_)

This function is used to change the value of **numberOfDimensions()** (§3.2.1). If the new value is different from the old value, then all geometric data is destroyed. It is important to be sure that the setting of **numberOfDimensions()** should be consistent with the topology of the mapping that defines the geometry of the grid.

3.4 Public data

3.4.1 AMR_ParentChildSiblingInfo* parentChildSiblingInfo

This is used for adaptive mesh refinement. Dan Quinlan++ has documentation for it.

3.5 Public data used only by derived classes

It is recommended that these variables be used only by derived classes.

3.5.1 MappedGridData* rcData

rcData is a pointer to the reference-counted data. It is recommended that this variable be used only by derived classes. See also the member functions **operator->()** (§3.2.53) and **operator*()** (§3.2.54), which are provided for access to **rcData**.

3.5.2 Logical isCounted

isCounted is a flag that indicates whether the data pointed to by **rcData** (§3.5.1) is known to be reference-counted. It is recommended that this variable be used only by derived classes.

3.6 Public constants

3.6.1 THEmask

THEmask indicates the discretization point **mask()** (§3.2.32). See also **update(what,how)** (§3.1.16) and **destroy(what)** (§3.1.18).

3.6.2 THEvertex

THEvertex indicates **vertex()** (§3.2.33), the locations of the vertices of the grid. See also **update(what,how)** (§3.1.16) and **destroy(what)** (§3.1.18).

3.6.3 THEcenter

THEcenter indicates **center()** (§3.2.34), the locations of the discretization points of the grid. See also **update(what,how)** (§3.1.16) and **destroy(what)** (§3.1.18).

3.6.4 THEcorner

THEcorner indicates **corner()** (§3.2.35), the locations of the discretization points of the grid. See also **update(what,how)** (§3.1.16) and **destroy(what)** (§3.1.18).

3.6.5 THEvertexDerivative

THEvertexDerivative indicates **vertexDerivative()** (§3.2.36), the derivative of the mapping at the vertices of the grid. See also **update(what,how)** (§3.1.16) and **destroy(what)** (§3.1.18).

3.6.6 THEcenterDerivative

THEcenterDerivative indicates **centerDerivative()** (§3.2.37), the derivative of the mapping at the discretization points of the grid. See also **update(what,how)** (§3.1.16) and **destroy(what)** (§3.1.18).

3.6.7 THEinverseVertexDerivative

THEinverseVertexDerivative indicates **inverseVertexDerivative()** (§3.2.38), the inverse of the mapping derivative, evaluated at the vertices of the grid. See also **update(what,how)** (§3.1.16) and **destroy(what)** (§3.1.18).

3.6.8 THEinverseCenterDerivative

THEinverseCenterDerivative indicates **inverseCenterDerivative()** (§3.2.39), the inverse of the mapping derivative, evaluated at the discretization points of the grid. See also **update(what,how)** (§3.1.16) and **destroy(what)** (§3.1.18).

3.6.9 THEvertexJacobian

THEvertexJacobian indicates **vertexJacobian()** (§3.2.40), the determinant of the derivative of the mapping at the vertices of the grid. See also **update(what,how)** (§3.1.16) and **destroy(what)** (§3.1.18).

3.6.10 THEcenterJacobian

THEcenterJacobian indicates **centerJacobian()** (§3.2.41), the determinant of the derivative of the mapping at the discretization points of the grid. See also **update(what,how)** (§3.1.16) and **destroy(what)** (§3.1.18).

3.6.11 THEcellVolume

THEcellVolume indicates **cellVolume()** (§3.2.42), the area (in two dimensions) or volume (in three dimensions) of the grid cells. See also **update(what,how)** (§3.1.16) and **destroy(what)** (§3.1.18).

3.6.12 THEfaceNormal

THEfaceNormal indicates **faceNormal()** (§3.2.43), the normals to the grid cell edges (in two dimensions) or faces (in three dimensions), normalized to the length or area area of the corresponding edge or face. See also **update(what,how)** (§3.1.16) and **destroy(what)** (§3.1.18).

3.6.13 THEcenterNormal

THEcenterNormal indicates **centerNormal()** (§3.2.44), the normals to constant parameter curves (in two dimensions) or surfaces (in three dimensions) passing through the grid cell centers, normalized to the length or area of that part of the curve or surface which lies inside the corresponding grid cell. See also **update(what,how)** (§3.1.16) and **destroy(what)** (§3.1.18).

3.6.14 THEfaceArea

THEfaceArea indicates **faceArea()** (§3.2.45), the length (in two dimensions) or area (in three dimensions) of the grid cell edges or faces. See also **update(what,how)** (§3.1.16) and **destroy(what)** (§3.1.18).

3.6.15 THEcenterArea

THEcenterArea indicates **centerArea()** (§3.2.46), the length (in two dimensions) or area (in three dimensions) of the grid cell edges or faces. See also **update(what,how)** (§3.1.16) and **destroy(what)** (§3.1.18).

3.6.16 THEvertexBoundaryNormal

THEvertexBoundaryNormal indicates **vertexBoundaryNormal** (§3.2.47), the unit outward normals to the grid boundary at the boundary vertices. See also **update(what,how)** (§3.1.16) and **destroy(what)** (§3.1.18).

3.6.17 THEcenterBoundaryNormal

THEcenterBoundaryNormal indicates **centerBoundaryNormal** (§3.2.48), the unit outward normals to the grid boundary at centers of the grid boundary cell edges (in two dimensions) or faces (in three dimensions). See also **update(what,how)** (§3.1.16) and **destroy(what)** (§3.1.18).

3.6.18 THEcenterBoundaryTangent

THEcenterBoundaryTangent indicates **centerBoundaryTangent** (§3.2.49), the unit tangent vectors to the grid boundary surface at centers of the grid boundary cell edges (in two dimensions) or faces (in three dimensions). See also **update(what,how)** (§3.1.16) and **destroy(what)** (§3.1.18).

3.6.19 THEminMaxEdgeLength

THEminMaxEdgeLength indicates **minimumEdgeLength()** (§3.2.28) and **maximumEdgeLength()** (§3.2.29), the minimum and maximum grid cell edge lengths. See also **update(what,how)** (§3.1.16) and **destroy(what)** (§3.1.18).

3.6.20 THEboundingBox

THEboundingBox indicates **boundingBox()** (§3.2.30), the coordinate bounds of a rectangular box that contains the vertices of the grid. See also **update(what,how)** (§3.1.16) and **destroy(what)** (§3.1.18).

3.6.21 THEusualSuspects

THEusualSuspects = **GenericGrid::THEusualSuspects** (§2.5.2) | **THEmask** (§3.6.1) | **THEvertex** (§3.6.2) | **THEcenter** (§3.6.3) | **THEvertexDerivative** (§3.6.5)
THEusualSuspects indicates some of the geometric data of a **MappedGrid**. The particular data indicated by **THEusualSuspects** may change from time to time. For this reason the use of **THEusualSuspects** is not recommended. **THEusualSuspects** overloads **GenericGrid::THEusualSuspects** (§2.5.2). See also **update(what,how)** (§3.1.16) and **destroy(what)** (§3.1.18).

3.6.22 EVERYTHING

EVERYTHING = **GenericGrid::EVERYTHING** (§2.5.3) | **THEmask** (§3.6.1) | **THEvertex** (§3.6.2) | **THEcenter** (§3.6.3) | **THEcorner** (§3.6.4) | **THEvertexDerivative** (§3.6.5) | **THEcenterDerivative** (§3.6.6) | **THEinverseVertexDerivative** (§3.6.7) | **THEinverseCenterDerivative** (§3.6.8) | **THEvertexJacobian** (§3.6.9) | **THEcenterJacobian** (§3.6.10) | **THEcellVolume** (§3.6.11) | **THEfaceNormal** (§3.6.12) | **THEcenterNormal** (§3.6.13) | **THEfaceArea** (§3.6.14) | **THEcenterArea** (§3.6.15) | **THEvertexBoundaryNormal** (§3.6.16) | **THEcenterBoundaryNormal** (§3.6.17) | **THEcenterBoundaryTangent** (§3.6.18) | **THEminMaxEdgeLength** (§3.6.19) | **THEboundingBox** (§3.6.20)
EVERYTHING indicates all of the geometric data associated with a **MappedGrid**. **EVERYTHING** overloads **GenericGrid::EVERYTHING** (§2.5.3). See also **update(what,how)** (§3.1.16) and **destroy(what)** (§3.1.18).

3.6.23 USEdifferenceApproximation

USEdifferenceApproximation indicates that computation of all geometric data except for **vertex()** (§3.2.33) should be done using discrete approximations such as finite-difference approximations. By default, if a **mapping** (§3.2.50) is available and is not of the base class "**Mapping**", then discrete approximations are not used. Instead, the mapping and its derivative are used to compute all of the geometric data. See also **update(what,how)** (§3.1.16).

3.6.24 COMPUTEgeometry

COMPUTEgeometry indicates that geometric data should be computed for each variable indicated, even if that data had already been computed and marked valid. See also **update(what,how)** (§3.1.16).

3.6.25 COMPUTEgeometryAsNeeded

COMPUTEgeometryAsNeeded indicates that geometric data should be computed only for those variables indicated, which either had not already been computed, were marked invalid, or for which new space needed to be allocated. See also **update(what,how)** (§3.1.16).

3.6.26 COMPUTEtheUsual

COMPUTEtheUsual = **GenericGrid::COMPUTEtheUsual** (§2.5.5) | **COMPUTEgeometryAsNeeded** (§3.6.25)
COMPUTEtheUsual indicates that computation of geometric data should proceed in the "usual way." Currently this means that geometric data is computed only for those variables indicated, which had not already been computed. However, this may change from time to time. **COMPUTEtheUsual** overloads **GenericGrid::COMPUTEtheUsual** (§2.5.5). See also **update(what,how)** (§3.1.16).

3.6.27 ISdiscretizationPoint

ISdiscretizationPoint indicates that a point is a discretization point.

3.6.28 ISinterpolationPoint

ISinterpolationPoint indicates that a point is an interpolation point.

3.6.29 ISghostPoint

ISghostPoint indicates a point outside the boundary, whose nearest boundary point is either a discretization point or an interpolation point.

3.6.30 ISinteriorBoundaryPoint

ISinteriorBoundaryPoint indicates that a point is a boundary point that lies in the interior of another grid, and should therefore not be used for the discretization of a boundary condition.

3.6.31 USESbackupRules

USESbackupRules indicates that a point uses backup rules for discretization or interpolation.

3.6.32 IShiddenByRefinement

IShiddenByRefinement indicates that a point is hidden by an overlying refinement grid.

3.6.33 ISreservedBit0

This constant should be used only by grid-generation programs.

3.6.34 ISreservedBit1

This constant should be used only by grid-generation programs.

3.6.35 ISreservedBit2

This constant should be used only by grid-generation programs.

3.6.36 GRIDnumberBits

This constant should be used only by grid-generation programs.

3.6.37 ISusedPoint

ISusedPoint = **ISdiscretizationPoint** (§3.6.27) | **ISinterpolationPoint** (§3.6.28) | **ISghostPoint** (§3.6.29)

4 Class GenericGridCollection

Class **GenericGridCollection** is the base class for all Overture classes that contain collections of grids. It contains a list of **GenericGrid**s. Each of the grids in this list may be considered to belong to a "base grid," which may have "refinement" grids at various levels of refinement. There may be more than one base grid, in which case the collection of grids may be partitioned into disjoint subsets of grids that belong to the various unrefined base grids. Class **GenericGridCollection** contains a list of **GenericGridCollection**s that may hold the subsets of grids that form this partition. It also contains a list of **GenericGridCollection**s that may hold the subsets of grids that have in common their level of refinement with respect to their base grids. This list forms a partition of the collection of grids into disjoint subsets according to refinement level. Each base grid or refinement may have more than one multigrid level. We call the finest multigrid level of any grid a "componenent grid," and the coarser multigrid levels of each component grid we call "multigrid coarsenings." There may be more than one component grid, in which case the collection of grids may be partitioned into disjoint subsets of grids that belong to the various component grids. Class **GenericGridCollection** contains a list of **GenericGridCollection**s that may hold the subsets of grids that form

this partition. It also contains a list of **GenericGridCollection**s that may hold the subsets of grids that have in common their level of multigrid coarsening with respect to their finest-level component grids. This list forms a partition of the collection of grids into disjoint subsets according to multigrid level. To summarize, the collection of grids may be partitioned according to base grid or refinement level, and these two partitions are dual to each other; similarly, the collection may be partitioned according to component grid or multigrid level, and these two partitions also are dual to each other. Since the lists forming the subsets in any of these four partitions are also **GenericGridCollection**s, they may be further partitioned in exactly the same way. For example, it is possible in this way to form a list of all the refinements of a particular base grid at a given multigrid level.

4.1 Public member functions

4.1.1 GenericGridCollection(const Integer numberOfGrids_ = 0)

Default constructor. If **numberOfGrids_**=0 (*e.g.*, by default) then create a null **GenericGridCollection**. Otherwise, create a **GenericGridCollection** with the given number of grids.

4.1.2 GenericGridCollection(const GenericGridCollection& x, const CopyType ct = DEEP)

Copy constructor. This does a deep copy by default. See also **operator=(x)** (§4.1.4) and **reference(x)** (§4.1.5).

4.1.3 virtual ~GenericGridCollection()

Destructor.

4.1.4 GenericGridCollection& operator=(const GenericGridCollection& x)

Assignment operator. This is also called a deep copy.

4.1.5 void reference(const GenericGridCollection& x)

Make a reference. This is also called a shallow copy. This **GenericGridCollection** shares the data of **x**.

4.1.6 virtual void breakReference()

Break a reference. If this **GenericGridCollection** shares data with any other **GenericGridCollection**, then this function replaces it with a new copy that does not share data.

4.1.7 virtual void consistencyCheck() const

Check the consistency of this **GenericGridCollection**.

4.1.8 virtual Integer get(const GenericDataBase& dir, const aString& name)

Copy a **GenericGridCollection** from a file.

4.1.9 virtual Integer put(GenericDataBase& dir, const aString& name) const

Copy a **GenericGridCollection** into a file.

4.1.10 Integer update(const Integer what = THEusualSuspects, const Integer how = COMPUTEtheUsual)

Update geometric data. The first argument (**what**) indicates which geometric data are to be updated. Any combination of the constants **THEbaseGrid** (§4.6.1), **THErefinementLevel** (§4.6.2), **THEcomponentGrid** (§4.6.3), **THEmultigridLevel** (§4.6.4), **NOTHING** (§4.6.5), **THEusualSuspects** (§4.6.6), **THElists** (§4.6.7) and **EVERYTHING** (§4.6.8), as well as any of the corresponding constants allowed for **GenericGrid::update(what,how)** (§2.1.10), may be bitwise ORed together to form the first argument of **update**(), to indicate which geometric data should be updated. This function returns a value obtained by bitwise ORing some of these constants, to indicate for which of the optional geometric data new array space was allocated. In addition, the constant **COMPUTEfailed** (§4.6.11), may be bitwise ORed into the value returned by **update**() in order to

indicate that the computation of some geometric data failed. The second argument (**how**) indicates whether and how any computation of geometric data should be done. Any combination of the constants **COMPUTEnothing** (§4.6.9), **COMPUTEtheUsual** (§4.6.10), as well as any of the corresponding constants allowed for **GenericGrid::update(what,how)** (§2.1.10), may be bitwise ORed together to form the optional second argument of **update**(). The corresponding function **update(what,how)** (§2.1.10) is called with the same arguments for each grid in the list **grid** (§4.4.1).

4.1.11 virtual Integer update(GenericGridCollection& x, const Integer what = THEusualSuspects, const Integer how = COMPUTEtheUsual)

Update geometric data, sharing space with the optional geometric data of another grid (**x**). If space for any indicated optional geometric data has not yet been allocated, or has the wrong dimensions, but **x** does contain the corresponding data, then the data for this **GenericGridCollection** will share space with the corresponding data of **x**. Any geometric data that already exists and has the correct dimensions is not forced to share space with the corresponding data of **x**. The corresponding function **update(x,what,how)** (§2.1.11) is called with the same arguments for each grid in the list **grid** (§4.4.1). For the optional arguments **what** and **how**, see the description of the function **update(what,how)** (§4.1.10).

4.1.12 void destroy(const Integer what = NOTHING)

Destroy the indicated optional geometric grid data. The argument (**what**) indicates which optional geometric data are to be destroyed. Any combination of the constants **THEbaseGrid** (§4.6.1), **THErefinementLevel** (§4.6.2), **THEcomponentGrid** (§4.6.3), **THEmultigridLevel** (§4.6.4), **NOTHING** (§4.6.5), **THEusualSuspects** (§4.6.6), **THElists** (§4.6.7) and **EVERYTHING** (§4.6.8), as well as any of the corresponding constants allowed for **GenericGrid::destroy(what)** (§2.1.12), may be bitwise ORed together to form the optional argument **what**. The corresponding function **destroy(what)** (§2.1.12) is called with the same argument for each grid in the list **grid** (§4.4.1).

4.1.13 void geometryHasChanged(const Integer what = ∼NOTHING)

Mark the geometric data out-of-date. Any combination of the constants **THEbaseGrid** (§4.6.1), **THErefinementLevel** (§4.6.2), **THEcomponentGrid** (§4.6.3), **THEmultigridLevel** (§4.6.4), **NOTHING** (§4.6.5), **THEusualSuspects** (§4.6.6), **THElists** (§4.6.7) and **EVERYTHING** (§4.6.8), as well as any of the corresponding constants allowed for **GenericGrid::geometryHasChanged(what)** (§2.1.13), may be bitwise ORed together to form the first argument of **geometryHasChanged**(). By default, all geometric data of this **GenericGridCollection** and all derived classes is marked out-of-date. The corresponding function **geometryHasChanged(what)** (§2.1.13) is called with the same argument for each grid in the list **grid** (§4.4.1). It is recommended that this function be called only from derived classes and grid-generation programs.

4.1.14 virtual Integer addRefinement(const Integer& level, const Integer k = 0)

Add a refinement grid to this collection. This refinement grid is marked as belonging to refinement level **level**. It is marked as belonging to the same base grid as that of **grid**[**k**], which may be any sibling, parent, or other more remote ancestor. It is required that **level** > 0. There must already exist grids marked as belonging to refinement level **level** $- 1$. This function returns the index of the refinement grid.

4.1.15 virtual void deleteRefinement(const Integer& k)

Delete all multigrid levels of refinement grid **k**.

4.1.16 virtual void deleteRefinementLevels(const Integer level = 0)

Delete all grids in this collection marked as belonging to refinement levels greater than **level**. It is required that **level** ≥ 0.

4.1.17 virtual void referenceRefinementLevels(GenericGridCollection& x, Integer level = INTEGER_MAX)

Make this collection contain exactly those grids from the collection **x** which are marked in **x** as belonging to refinement level **level** or to any coarser refinent level. It is required that **level** ≥ 0.

4.1.18 virtual Integer addMultigridCoarsening(const Integer& level, const Integer k = 0)

Add a multigrid coarsening of a grid to this collection. This coarsening is marked as belonging to multigrid level **level**. It is marked as belonging to the same component grid as that of **grid**[**k**], which may be any finer multigrid level of the same component grid. It is required that **level** > 0. There must already exist a multigrid coarsening of the component grid at multigrid level **level** $- 1$. This function returns the index of the multigrid coarsening.

4.1.19 virtual void deleteMultigridCoarsening(const Integer& k)

Delete grid **k**,a multigrid coarsening, and all of its coarser multigrid levels.

4.1.20 virtual void deleteMultigridLevels(const Integer level = 0)

Delete all grids in this collection marked as belonging to multigrid levels greater than **level**. It is required that **level** ≥ 0.

4.1.21 virtual void initialize(const Integer& numberOfGrids_)

Initialize the GenericGridCollection with the given number of grids. These grids have their gridNumbers, baseGridNumbers and componentGridNumbers set to $[0, \ldots,$ numberOfGrids_ $- 1]$, and their refinementLevelNumbers and multigridLevelNumbers set to zero.

4.1.22 Logical operator==(const GenericGridCollection& x) const

This comparison function returns **LogicalTrue** (non-zero) if and only if **x** refers the same grid as $*$**this**.

4.1.23 Logical operator!=(const GenericGridCollection& x) const

This comparison function returns **LogicalTrue** or (non-zero) if and only if **x** does not refer to the same grid as $*$**this**.

4.1.24 Integer getIndex(const GenericGrid& x) const

This function returns the index of grid **x** in $*$**this**, or returns -1 if **x** is not in $*$**this**.

4.2 Public Member functions for access to data

4.2.1 const Integer& computedGeometry() const

This function returns a reference to a bit mask that indicates which geometrical data has been computed. This mask must be reset to zero to invalidate the data when the geometry changes. It is recommended that this data be used only by derived classes and grid-generation programs. See also **geometryHasChanged(what)** (§4.1.13).

4.2.2 const Integer& numberOfGrids() const

This function returns a reference to the number of grids in the list **grid** (§4.4.1).

4.2.3 const Integer& numberOfBaseGrids() const

This function returns a reference to the number of **GenericGridCollection**s in the list **baseGrid** (§4.4.3).

4.2.4 const Integer& numberOfRefinementLevels() const

This function returns a reference to the number of **GenericGridCollection**s in the list **refinementLevel** (§4.4.5).

4.2.5 const Integer& numberOfComponentGrids() const

This function returns a reference to the number of **GenericGridCollection**s in the list **multigridLevel** (§4.4.9).

4.2.6 const Integer& numberOfMultigridLevels() const

This function returns a reference to the number of **GenericGridCollection**s in the list **multigridLevel** (§4.4.9).

4.2.7 GenericGrid& operator[](const int& i)

Get a reference to the ith grid. If **g** is a **GenericGridCollection**, then **g**$[i]$ is a reference to the **GenericGrid g**.**grid**$[i]$.

4.2.8 virtual aString getClassName() const

Get the class name of the most-derived class for this object.

4.2.9 GenericGridCollectionData* operator->()

Access the reference-counted data.

4.2.10 GenericGridCollectionData& operator*()

Access the pointer to the reference-counted data.

4.3 Public member functions called only from derived classes

It is recommended that these functions be called only from derived classes.

4.3.1 void reference(GenericGridCollectionData& x)

Make a reference to an object of type **GenericGridCollectionData**. This **GenericGridCollection** uses **x** for its data. It is recommended that this function be called only from derived classes.

4.3.2 void updateReferences(const Integer what = EVERYTHING)

Update references to the reference-counted data. It is recommended that this function be called only from derived classes.

4.3.3 virtual void setNumberOfGrids(const Integer& numberOfGrids_)

Add or delete grids to/from this collection until there are **numberOfGrids_** grids. Any grids added are marked as belonging to base grid zero, multigrid level zero and refinement level zero. The use of this function is not recommended if the collection is intended to contain more than one base grid, refinement level, component grid or multigrid level. For example, if the collection is intended to contain more than one refinement level, it is recommended that the functions **addRefinement(level,k)** (§4.1.14) and **deleteRefinementLevels(level)** (§4.1.16) be used instead.

4.4 Public data

4.4.1 ListOfGenericGrid grid

Length: **numberOfGrids()** (§4.2.2)
A list containing of all of the grids in the collection. The grids in this list are **GenericGrid**s.

4.4.2 const IntegerArray gridNumber

Dimensions: $(0{:}\, n_1)$, where $n_1 =$ **numberOfGrids**$() - 1$.
The index of the each **GenericGrid** in the list **grid** (§4.4.1).

4.4.3 ListOfGenericGridCollection baseGrid

baseGrid is a list of **GenericGridCollection**s containing one base grid and all of its refinements, including all multigrid coarsenings of these grids. This data may be updated as in the following example.

4.4.4 const IntegerArray baseGridNumber

Dimensions: $(0{:}\, n_1)$, where $n_1 =$ **numberOfGrids**$() - 1$.
baseGridNumber holds the index of the base grid ancestor of each **GenericGrid** in the list **grid** (§4.4.1).

4.4.5 ListOfGenericGridCollection refinementLevel

refinementLevel is a list of **GenericGridCollection**s containing **GenericGrid**s that belong to the same refinement level. This data may be updated as in the following example.

4.4.6 const IntegerArray refinementLevelNumber

Dimensions: $(0{:}\, n_1)$, where $n_1 =$ **numberOfGrids**$() - 1$.
refinementLevelNumber holds the index of the refinement level of each **GenericGrid** in the list **grid** (§4.4.1).

4.4.7 ListOfGenericGridCollection componentGrid

componentGrid is a list of **GenericGridCollection**s containing one componenent grid and all of its multigrid coarsenings. This data may be updated as in the following example.

4.4.8 const IntegerArray componentGridNumber

Dimensions: $(0{:}\, n_1)$, where $n_1 =$ **numberOfGrids**$() - 1$.
componentGridNumber holds the index of the component grid of each **GenericGrid** in the list **grid** (§4.4.1).

4.4.9 ListOfGenericGridCollection multigridLevel

multigridLevel is a list of **GenericGridCollection**s containing **GenericGrid**s that belong to the same multigrid level. This data may be updated as in the following example.

4.4.10 const IntegerArray multigridLevelNumber

Dimensions: $(0{:}\, n_1)$, where $n_1 =$ **numberOfGrids**$() - 1$.
multigridLevelNumber holds the index of the multigrid level of each **GenericGrid** in the list **grid** (§4.4.1).

4.5 Public data used only by derived classes

It is recommended that these variables be used only by derived classes.

4.5.1 GenericGridCollectionData* rcData

rcData is a pointer to the reference-counted data. It is recommended that this variable be used only by derived classes. See also the member functions **operator->()** (§4.2.9) and **operator*()** (§4.2.10), which are provided for access to **rcData**.

4.5.2 Logical isCounted

isCounted is a flag that indicates whether the data pointed to by **rcData** (§4.5.1) is known to be reference-counted. It is recommended that this variable be used only by derived classes.

4.6 Public constants

4.6.1 THEbaseGrid

THEbaseGrid indicates **baseGrid** (§4.4.3), the list of **GenericGridCollection**s containing those **GenericGrid**s which are descended from the same base grid. See also **update(what,how)** (§4.1.10) and **destroy(what)** (§4.1.12).

4.6.2 THErefinementLevel

THErefinementLevel indicates **refinementLevel** (§4.4.5), the list of **GenericGridCollection**s containing those **GenericGrid**s which belong to the same refinement level. See also **update(what,how)** (§4.1.10) and **destroy(what)** (§4.1.12).

4.6.3 THEcomponentGrid

THEcomponentGrid indicates **multigridLevel** (§4.4.7), the list of **GenericGridCollection**s containing those **GenericGrid**s which belong to the same component grid. See also **update(what,how)** (§4.1.10) and **destroy(what)** (§4.1.12).

4.6.4 THEmultigridLevel

THEmultigridLevel indicates **multigridLevel** (§4.4.9), the list of **GenericGridCollection**s containing those **GenericGrid**s which belong to the same multigrid level. See also **update(what,how)** (§4.1.10) and **destroy(what)** (§4.1.12).

4.6.5 NOTHING

NOTHING = **GenericGrid::NOTHING** (§2.5.1) See also **update(what,how)** (§4.1.10) and **destroy(what)** (§4.1.12).

4.6.6 THEusualSuspects

THEusualSuspects = **GenericGrid::THEusualSuspects** (§2.5.2)
THEusualSuspects indicates some of the geometric data of a **GenericGridCollection**. The particular data indicated by **THEusualSuspects** may change from time to time. For this reason the use of **THEusualSuspects** is not recommended. See also **update(what,how)** (§4.1.10) and **destroy(what)** (§4.1.12).

4.6.7 THElists

THElists = **THEbaseGrid** (§4.6.1) | **THErefinementLevel** (§4.6.2) | **THEcomponentGrid** (§4.6.3) | **THEmultigridLevel** (§4.6.4)
See also **update(what,how)** (§4.1.10) and **destroy(what)** (§4.1.12).

4.6.8 EVERYTHING

EVERYTHING = **GenericGrid::EVERYTHING** (§2.5.3) | **THEbaseGrid** (§4.6.1) | **THEmultigridLevel** (§4.6.4) | **THErefinementLevel** (§4.6.2)
EVERYTHING indicates all of the geometric data associated with a **GenericGridCollection**. See also **update(what,how)** (§4.1.10) and **destroy(what)** (§4.1.12).

4.6.9 COMPUTEnothing

COMPUTEnothing = **GenericGrid::COMPUTEnothing** (§2.5.4)
See also **update(what,how)** (§4.1.10) and **destroy(what)** (§4.1.12).

4.6.10 COMPUTEtheUsual

COMPUTEtheUsual = **GenericGrid::COMPUTEtheUsual** (§2.5.5)
See also **update(what,how)** (§4.1.10) and **destroy(what)** (§4.1.12).

4.6.11 COMPUTEfailed

COMPUTEfailed = **GenericGrid::COMPUTEfailed** (§2.5.6)
See also **update(what,how)** (§4.1.10) and **destroy(what)** (§4.1.12).

5 Class GridCollection

Class **GridCollection** is the base class for all Overture classes that contain collections of **MappedGrid**s. It is derived from class **GenericGridCollection**.

5.1 Public member functions

5.1.1 GridCollection(const Integer numberOfDimensions_ = 0, const Integer numberOfGrids_ = 0)

Default constructor. If numberOfDimensions_==0 (*e.g.*, by default) then create a null **GridCollection**. Otherwise, create a **GridCollection** with the given number of dimensions and number of grids.

5.1.2 GridCollection(const GridCollection& x, const CopyType ct = DEEP)

Copy constructor. This does a deep copy by default. See also **operator=(x)** (§5.1.4) and **reference(x)** (§5.1.5).

5.1.3 virtual ~GridCollection()

Destructor.

5.1.4 GridCollection& operator=(const GridCollection& x)

Assignment operator. This is also called a deep copy.

5.1.5 void reference(const GridCollection& x)

Make a reference. This is also called a shallow copy. This **GridCollection** shares the data of **x**.

5.1.6 virtual void breakReference()

Break a reference. If this **MappedGrid** shares data with any other **GridCollection**, then this function replaces it with a new copy that does not share data.

5.1.7 void changeToAllVertexCentered()

Change the grid to be all vertex-centered.

5.1.8 void changeToAllCellCentered()

Change the grid to be all cell-centered.

5.1.9 virtual void consistencyCheck() const

Check the consistency of this **GridCollection**.

5.1.10 virtual Integer get(const GenericDataBase& dir, const aString& name)

Copy a **GridCollection** from a file.

5.1.11 virtual Integer put(GenericDataBase& dir, const aString& name) const

Copy a **GridCollection** into a file.

5.1.12 Integer update(const Integer what = THEusualSuspects, const Integer how = COMPUTEtheUsual)

Update geometric data. The first argument (**what**) indicates which geometric data are to be updated. Any combination of the constants **THEusualSuspects** (§5.6.21) and **EVERYTHING** (§5.6.22), as well as any of the corresponding constants allowed for **GenericGridCollection::update(what,how)** (§4.1.10) or **MappedGrid::update(what,how)** (§3.1.16), may be bitwise ORed together to form the first argument of **update**(), to indicate which geometric data should be updated. This function returns a value obtained by bitwise ORing some of these constants, to indicate for which of the optional geometric data new array space was allocated. In addition, the constant **COMPUTEfailed** (§4.6.11), may be bitwise ORed into the value returned by **update**() in order to indicate that the computation of some geometric data failed. The second argument (**how**) indicates whether and how any computation of geometric data should be done. The constant **COMPUTEtheUsual** (§5.6.26), as well as any of the corresponding constants allowed for **GenericGridCollection::update(what,how)** (§4.1.10) or **MappedGrid::update(what,how)** (§3.1.16), may be bitwise ORed together to form the optional second argument of **update**(). The corresponding function **update(what,how)** (§4.1.10) is called with the same arguments for the base class **GenericGridCollection**, and **update(what,how)** (§3.1.16) is called with the same arguments for each **MappedGrid** in the list **grid** (§5.4.4).

5.1.13 Integer update(GridCollection& x, const Integer what = THEusualSuspects, const Integer how = COMPUTEtheUsual)

Update geometric data, sharing space with the optional geometric data of another grid (**x**). If space for any indicated optional geometric data has not yet been allocated, or has the wrong dimensions, but **x** does contain the corresponding data, then the data for this **GridCollection** will share space with the corresponding data of **x**. Any geometric data that already exists and has the correct dimensions is not forced to share space with the corresponding data of **x**. The corresponding function **update(x,what,how)** (§4.1.11) is called with the same arguments for the base class **GenericGridCollection**, and **update(what,how)** (§3.1.16) is called with the same arguments for each **MappedGrid** in the list **grid** (§5.4.4). For the optional arguments **what** and **how**, see the description of the function **update(what,how)** (§5.1.12).

5.1.14 virtual void destroy(const Integer what = NOTHING)

Destroy the indicated optional geometric grid data. The argument (**what**) indicates which optional geometric data are to be destroyed. Any combination of the constants **THEusualSuspects** (§5.6.21) and **EVERYTHING** (§5.6.22), as well as any of the corresponding constants allowed for **GenericGridCollection::destroy(what)** (§4.1.12) or **MappedGrid::destroy(what)** (§3.1.18), may be bitwise ORed together to form the optional argument **what**. The corresponding function **destroy(what)** (§4.1.12) is called with the same argument for the base class **GenericGridCollection**, and **destroy(what)** (§3.1.18) is called with the same argument for each grid in the list **grid** (§5.4.4).

5.1.15 virtual Integer addRefinement(const IntegerArray& range, const IntegerArray& factor) const Integer& level, const Integer k = 0)

Add a refinement grid to this collection. This refinement grid is marked as belonging to refinement level **level**. It is marked as having the same base grid as that of **grid**[**k**], which may be any sibling, parent, or other more remote ancestor. It is required that **level** > 0. There must already exist grids marked as belonging to refinement level **level** $- 1$. The **indexRange()** (§3.2.3) of the refinement is computed from **range**, which must be a subset of the index range of a refinement of the same base grid (the refinement grid's most remote ancestor), at the next-coarser level of refinement (the refinement level of any parent grid), that would cover the entire base grid. The refinement factor **factor** is relative to any parent grid; any cell of any parent grid that is covered by cells of the new refinement grid is subdivided in each direction k into **factor**(k) cells of the new refinement grid. This function returns the index of the refinement grid.

5.1.16 Integer addRefinement(const IntegerArray& range, const Integer& factor) const Integer& level, const Integer k = 0)

Add a refinement grid to this collection. This function simply calls **addRefinement()** (§5.1.15) with **factor** replaced by an **IntegerArray** of length three set to the scalar **factor**. This function returns the index of the refinement grid.

5.1.17 virtual void deleteRefinement(const Integer& k)

Delete all multigrid levels of refinement grid **k**.

5.1.18 virtual void deleteRefinementLevels(const Integer& level)

Delete all grids in this collection marked as belonging to refinement levels greater than **level**. It is required that **level** ≥ 0.

5.1.19 void referenceRefinementLevels(GridCollection& x, Integer level = INTEGER_MAX)

Make this collection contain exactly those grids from the collection **x** which are marked in **x** as belonging to refinement level **level** or to any coarser refinent level. It is required that **level** ≥ 0.

5.1.20 virtual Integer addMultigridCoarsening(const IntegerArray& factor, const Integer& level, const Integer k = 0)

Add a multigrid coarsening of a grid to this collection. The multigrid coarsening factor in each index direction, relative to the next-finer multigrid level, is given by **factor**. This coarsening is marked as belonging to multigrid level **level**. It is marked as belonging to the same component grid as that of **grid**[**k**], which may be any finer multigrid level of the same component grid. It is required that **level** > 0. There must already exist a multigrid coarsening of the component grid at multigrid level **level** $- 1$. This function returns the index of the multigrid coarsening.

5.1.21 Integer addMultigridCoarsening(const Integer& factor, const Integer& level, const Integer k = 0)

Add a multigrid coarsening of a grid to this collection. The multigrid coarsening factor relative to the next-finer multigrid level is given by **factor**. This coarsening is marked as belonging to multigrid level **level**. It is marked as belonging to the same component grid as that of **grid**[**k**], which may be any finer multigrid level of the same component grid. It is required that **level** > 0. There must already exist a multigrid coarsening of the component grid at multigrid level **level** $- 1$. This function returns the index of the multigrid coarsening.

5.1.22 virtual void deleteMultigridCoarsening(const Integer& k)

Delete grid **k**,a multigrid coarsening, and all of its coarser multigrid levels.

5.1.23 virtual void deleteMultigridLevels(const Integer level = 0)

Delete all grids in this collection marked as belonging to multigrid levels greater than **level**. It is required that **level** ≥ 0.

5.1.24 virtual void initialize(const Integer& numberOfDimensions_, Integer& numberOfGrids_)

Initialize the GridCollection with the given number of grids. These grids have their gridNumbers, baseGridNumbers and componentGridNumbers set to $[0, \ldots,$ numberOfGrids_ $- 1]$, and their refinementLevelNumbers and multigridLevelNumbers set to zero.

5.2 Public Member functions for access to data

5.2.1 const Integer& numberOfDimensions() const

This function returns a reference to the number of dimensions of the grids in the list **grid** (§4.4.1).

5.2.2 virtual aString getClassName() const

Get the class name of the most-derived class for this object.

5.2.3 MappedGrid& operator[](const int& i) const

Get a reference to the ith grid. If **g** is a **GridCollection**, then **g**$[i]$ is a reference to the **MappedGrid g**.**grid**$[i]$.

5.2.4 GridCollectionData* operator->()

Access the reference-counted data.

5.2.5 GridCollectionData& operator*()

Access the pointer to the reference-counted data.

5.3 Public member functions called only from derived classes

It is recommended that these functions be called only from derived classes.

5.3.1 void reference(GridCollectionData& x)

Make a reference to an object of type **GridCollectionData**. This **GridCollection** uses **x** for its data. It is recommended that this function be called only from derived classes.

5.3.2 void updateReferences(const Integer what = EVERYTHING)

Update references to the reference-counted data. It is recommended that this function be called only from derived classes.

5.3.3 virtual void setNumberOfGrids(const Integer& numberOfGrids_)

Set the number of grids, and add or delete grids to/from this collection until there are **numberOfGrids_** grids. See also **setNumberOfDimensionsAndGrids(numberOfDimensions_, numberOfGrids_)** (§5.3.5).

5.3.4 virtual void setNumberOfDimensions(const Integer& numberOfDimensions_)

Set the number of dimensions. This is valid only if the collection does not contain **MappedGrid**s with a different number of dimensions. See also **setNumberOfDimensionsAndGrids(numberOfDimensions_, numberOfGrids_)** (§5.3.5).

5.3.5 virtual void setNumberOfDimensionsAndGrids(const Integer& numberOfDimensios_, const Integer& numberOfGrids_)

Set the number of dimensions of the grids, and add or delete grids to/from this collection until there are **numberOfGrids_** grids. Any grids added are marked as belonging to base grid zero, multigrid level zero and refinement level zero. The use of this function is not recommended if the collection is intended to contain more than one base grid, multigrid level or refinement level. For example, if the collection is intended to contain more than one refinement level, it is recommended that the functions **addRefinement(range,factor,level,k)** (§5.1.15) and **deleteRefinementLevels(level)** (§5.1.18) be used instead.

5.4 Public data

5.4.1 const RealArray boundingBox

Dimensions: $(0{:}\,2)$
boundingBox holds coordinate bounds for the grids in the list **grid** (§5.4.4). It is computed as the min/max of **MappedGrid::boundingBox()** (§3.2.30) from each grid in the collection.

5.4.2 const IntegerArray refinementFactor

Dimensions: $(0{:}\,2, 0{:}\,\textbf{numberOfGrids}() - 1)$
refinementFactor holds the ratio, for each grid, in each index direction, of the size of cells of the corresponding unrefined base grid (at the same multigrid level) to the size of cells of that grid.

5.4.3 const IntegerArray multigridCoarseningFactor

Dimensions: $(0{:}\,2, 0{:}\,\textbf{numberOfGrids}() - 1)$
multigridCoarseningFactor holds the ratio, for each grid, in each index direction, of the size of cells of that grid to the size of cells of the finest multigrid level of the same component grid.

5.4.4 ListOfMappedGrid grid

Length: **numberOfGrids()** (§4.2.2)
A list containing of all of the grids in the collection. The grids in this list are **MappedGrid**s. **grid** overloads **GenericGridCollection::grid** (§4.4.1), which contains the same **MappedGrid**s (referred to in the latter case as **GenericGrid**s).

5.4.5 ListOfGridCollection baseGrid

baseGrid is a list of **GridCollection**s containing one base grid and all of its refinements, including all multigrid coarsenings of these grids. **baseGrid** overloads **GenericGridCollection::baseGrid** (§4.4.3), which contains the same **GridCollection**s (referred to in the latter case as **GenericGridCollection**s). This data may be updated as in the following example.

5.4.6 ListOfGridCollection refinementLevel

refinementLevel is a list of **GridCollection**s containing those **MappedGrid**s which belong to the same refinement level. **refinementLevel** overloads **GenericGridCollection::refinementLevel** (§4.4.5), whose **GenericGridCollection**s contain the same grids as the **GridCollection**s of **refinementLevel**. This data may be updated as in the following example.

5.4.7 ListOfGridCollection componentGrid

componentGrid is a list of **GridCollection**s containing one componenent grid and all of its multigrid coarsenings. **componentGrid** overloads **GenericGridCollection::componentGrid** (§4.4.7), which contains the same **GridCollection**s (referred to in the latter case as **GenericGridCollection**s). This data may be updated as in the following example.

5.4.8 ListOfGridCollection multigridLevel

multigridLevel is a list of **GridCollection**s containing those **MappedGrid**s which belong to the same multigrid level. **multigridLevel** overloads **GenericGridCollection::multigridLevel** (§4.4.9), whose **GenericGridCollection**s contain the same grids as the **GridCollection**s of **multigridLevel** (referred to in the latter case as **GenericGrid**s). This data may be updated as in the following example.

5.4.9 AMR_RefinementLevelInfo* refinementLevelInfo

This is used for adaptive mesh refinement. Dan Quinlan++ has documentation for it.

5.5 Public data used only by derived classes

It is recommended that these variables be used only by derived classes.

5.5.1 GridCollectionData* rcData

rcData is a pointer to the reference-counted data. It is recommended that this variable be used only by derived classes. See also the member functions **operator->()** (§5.2.4) and **operator*()** (§5.2.5), which are provided for access to **rcData**.

5.5.2 Logical isCounted

isCounted is a flag that indicates whether the data pointed to by **rcData** (§5.5.1) is known to be reference-counted. It is recommended that this variable be used only by derived classes.

5.6 Public constants

Class **GridCollection** has all of the same constants defined as classes **MappedGrid** and **GenericGridCollection**. These are listed here for the sake of completeness.

5.6.1 THEmask

THEmask = **MappedGrid::THEmask** (§3.6.1)

5.6.2 THEvertex

THEvertex = **MappedGrid::THEvertex** (§3.6.2)

5.6.3 THEcenter

THEcenter = **MappedGrid::THEcenter** (§3.6.3)

5.6.4 THEcorner

THEcorner = **MappedGrid::THEcorner** (§3.6.4)

5.6.5 THEvertexDerivative

THEvertexDerivative = **MappedGrid::THEvertexDerivative** (§3.6.5)

5.6.6 THEcenterDerivative

THEcenterDerivative = **MappedGrid::THEcenterDerivative** (§3.6.6)

5.6.7 THEinverseVertexDerivative

THEinverseVertexDerivative = **MappedGrid::THEinverseVertexDerivative** (§3.6.7)

5.6.8 THEinverseCenterDerivative

THEinverseCenterDerivative = **MappedGrid::THEinverseCenterDerivative** (§3.6.8)

5.6.9 THEvertexJacobian

THEvertexJacobian = **MappedGrid::THEvertexJacobian** (§3.6.9)

5.6.10 THEcenterJacobian

THEcenterJacobian = **MappedGrid::THEcenterJacobian** (§3.6.10)

5.6.11 THEcellVolume

THEcellVolume = **MappedGrid::THEcellVolume** (§3.6.11)

5.6.12 THEfaceNormal

THEfaceNormal = **MappedGrid::THEfaceNormal** (§3.6.12)

5.6.13 THEcenterNormal

THEcenterNormal = **MappedGrid::THEcenterNormal** (§3.6.13)

5.6.14 THEfaceArea

THEfaceArea = **MappedGrid::THEfaceArea** (§3.6.14)

5.6.15 THEcenterArea

THEcenterArea = **MappedGrid::THEcenterArea** (§3.6.15)

5.6.16 THEvertexBoundaryNormal

THEvertexBoundaryNormal = **MappedGrid::THEvertexBoundaryNormal** (§3.6.16)

5.6.17 THEcenterBoundaryNormal

THEcenterBoundaryNormal = **MappedGrid::THEcenterBoundaryNormal** (§3.6.17)

5.6.18 THEcenterBoundaryTangent

THEcenterBoundaryTangent = **MappedGrid::THEcenterBoundaryTangent** (§3.6.18)

5.6.19 THEminMaxEdgeLength

THEminMaxEdgeLength = **MappedGrid::THEminMaxEdgeLength** (§3.6.19)

5.6.20 THEboundingBox

THEboundingBox = **MappedGrid::THEboundingBox** (§3.6.20)

5.6.21 THEusualSuspects

THEusualSuspects = **GenericGridCollection::THEusualSuspects** (§4.6.6) | **MappedGrid::THEusualSuspects** (§3.6.21)
THEusualSuspects indicates some of the geometric data of a **GridCollection**. The particular data indicated by **THEusualSuspects** may change from time to time. For this reason the use of **THEusualSuspects** is not recommended. See also **update(what,how)** (§5.1.12) and **destroy(what)** (§5.1.14).

5.6.22 EVERYTHING

EVERYTHING = **GenericGridCollection::EVERYTHING** (§4.6.8) | **MappedGrid::EVERYTHING** (§3.6.22)
EVERYTHING indicates all of the geometric data associated with a **GridCollection**. (overloaded) See also **update(what,how)** (§5.1.12) and **destroy(what)** (§5.1.14).

5.6.23 USEdifferenceApproximation

USEdifferenceApproximation = **MappedGrid::USEdifferenceApproximation** (§3.6.23)
See also **update(what,how)** (§5.1.12) and **destroy(what)** (§5.1.14).

5.6.24 COMPUTEgeometry

COMPUTEgeometry = **MappedGrid::COMPUTEgeometry** (§3.6.24)
See also **update(what,how)** (§5.1.12) and **destroy(what)** (§5.1.14).

5.6.25 COMPUTEgeometryAsNeeded

COMPUTEgeometryAsNeeded = **MappedGrid::COMPUTEgeometryAsNeeded** (§3.6.25)
See also **update(what,how)** (§5.1.12) and **destroy(what)** (§5.1.14).

5.6.26 COMPUTEtheUsual

COMPUTEtheUsual = **GenericGridCollection::COMPUTEtheUsual** (§4.6.10) | **MappedGrid::COMPUTEtheUsual** (§3.6.26)
See also **update(what,how)** (§5.1.12) and **destroy(what)** (§5.1.14).

5.6.27 ISdiscretizationPoint

ISdiscretizationPoint = **MappedGrid::ISdiscretizationPoint** (§3.6.27)

5.6.28 ISinterpolationPoint

ISinterpolationPoint = **MappedGrid::ISinterpolationPoint** (§3.6.28)

5.6.29 ISghostPoint

ISghostPoint = **MappedGrid::ISghostPoint** (§3.6.29)

5.6.30 ISinteriorBoundaryPoint

ISinteriorBoundaryPoint = **MappedGrid::ISinteriorBoundaryPoint** (§3.6.30)

5.6.31 USESbackupRules

USESbackupRules = **MappedGrid::USESbackupRules** (§3.6.31)

5.6.32 IShiddenByRefinement

IShiddenByRefinement = **MappedGrid::IShiddenByRefinement** (§3.6.32)

5.6.33 ISreservedBit0

ISreservedBit0 = **MappedGrid::ISreservedBit0** (§3.6.33)

5.6.34 ISreservedBit1

ISreservedBit1 = **MappedGrid::ISreservedBit1** (§3.6.34)

5.6.35 ISreservedBit2

ISreservedBit2 = **MappedGrid::ISreservedBit2** (§3.6.35)

5.6.36 GRIDnumberBits

GRIDnumberBits = **MappedGrid::GRIDnumberBits** (§3.6.36)

5.6.37 ISusedPoint

ISusedPoint = **MappedGrid::ISusedPoint** (§3.6.37)

6 Class CompositeGrid

Class **CompositeGrid** is used for a composite overlapping grid, which is a collection of **MappedGrid**s and a description of how function values defined on these grids are related through interpolation between grids in their regions of overlap. Class **CompositeGrid** is derived from class **GridCollection**.

6.1 Public member functions

6.1.1 CompositeGrid(const Integer numberOfDimensions_ = 0, const Integer numberOfComponentGrids_ = 0)

Default constructor. If numberOfDimensions_==0 (*e.g.*, by default) then create a null CompositeGrid. Otherwise create a CompositeGrid with the given number of dimensions and number of component grids.

6.1.2 CompositeGrid(const CompositeGrid& x, const CopyType ct = DEEP)

Copy constructor. This does a deep copy by default. See also **operator=(x)** (§6.1.4) and **reference(x)** (§6.1.5).

6.1.3 virtual ~CompositeGrid()

Destructor.

6.1.4 CompositeGrid& operator=(const CompositeGrid& x)

Assignment operator. This is also called a deep copy.

6.1.5 void reference(const CompositeGrid& x)

Make a reference. This is also called a shallow copy. This **CompositeGrid** shares the data of **x**.

6.1.6 virtual void breakReference()

Break a reference. If this **CompositeGrid** shares data with any other **CompositeGrid**, then this function replaces it with a new copy that does not share data.

6.1.7 void changeToAllVertexCentered()

Change the **MappedGrid**s in this **CompositeGrid** to be all vertex-centered.

6.1.8 void changeToAllCellCentered()

Change the **MappedGrid**s in this **CompositeGrid** to be all cell-centered.

6.1.9 virtual void consistencyCheck() const

Check the consistency of this **CompositeGrid**.

6.1.10 virtual Integer get(const GenericDataBase& dir, const aString& name)

Copy a **CompositeGrid** from a file.

6.1.11 virtual Integer put(GenericDataBase& dir, const aString& name) const

Copy a **CompositeGrid** into a file.

6.1.12 Integer update(const Integer what = THEusualSuspects, const Integer how = COMPUTEtheUsual)

Update geometric data. The first argument (**what**) indicates which geometric data are to be updated. Any combination of the constants **THEinterpolationCoordinates** (§6.7.1), **THEinterpoleeGrid** (§6.7.2), **THEinterpoleeLocation** (§6.7.3), **THEinterpolationPoint** (§6.7.4), **THEinterpolationCondition** (§6.7.5), **THEinverseMap** (§6.7.6), **THEusualSuspects** (§6.7.7) and **EVERYTHING** (§6.7.8), as well as any of the corresponding constants allowed for **GenericGridCollection::update(what,how)** (§4.1.10), may be bitwise ORed together to form the first argument of **update**(), to indicate which geometric data should be updated. This function returns a value obtained by bitwise ORing some of these constants, to indicate for which of the optional geometric data new array space was allocated. In addition, the constant **COMPUTEfailed** (§4.6.11), may be bitwise ORed into the value returned by **update**() in order to indicate that the computation of some geometric data failed. The second argument (**how**) indicates whether and how any computation of geometric data should be done. The constant **COMPUTEtheUsual** (§6.7.9), as well as any of the corresponding constants allowed for **GenericGridCollection::update(what,how)** (§4.1.10), may be bitwise ORed together to form the optional second argument of **update**(). The corresponding function **update(what,how)** (§5.1.12) is called with the same arguments for the base class **GridCollection**.

6.1.13 Integer update(CompositeGrid& x, const Integer what = THEusualSuspects, const Integer how = COMPUTEtheUsual)

Update geometric data, sharing space with the optional geometric data of another grid (**x**). If space for any indicated optional geometric data has not yet been allocated, or has the wrong dimensions, but **x** does contain the corresponding data, then the data for this **CompositeGrid** will share space with the corresponding data of **x**. Any geometric data that already exists and has the correct dimensions is not forced to share space with the corresponding data of **x**. The corresponding function **update(x,what,how)** (§5.1.13) is called with the same arguments for the base class **GenericGridCollection**. For the optional arguments **what** and **how**, see the description of the function **update(what,how)** (§6.1.12).

6.1.14 virtual void destroy(const Integer what = NOTHING)

Destroy the indicated optional geometric grid data. The argument (**what**) indicates which optional geometric data are to be destroyed. Any combination of the constants **THEinterpolationCoordinates** (§6.7.1), **THEinterpoleeGrid** (§6.7.2), **THEinterpoleeLocation** (§6.7.3), **THEinterpolationPoint** (§6.7.4), **THEinterpolationCondition** (§6.7.5), **THEinverseMap** (§6.7.6), **THEusualSuspects** (§6.7.7) and **EVERYTHING** (§6.7.8), as well as any of the corresponding constants allowed for **GenericGridCollection::destroy(what)** (§4.1.12) may be bitwise ORed together to form the optional argument **what**. The corresponding function **destroy(what)** (§5.1.14) is called with the same argument for the base class **GridCollection**.

6.1.15 virtual Integer addRefinement(const IntegerArray& range, const IntegerArray& factor, const Integer& level, const Integer k = 0)

Add a refinement grid to this collection. This refinement grid is marked as belonging to refinement level **level**. It is marked as having the same base grid as that of **grid**[**k**], which may be any sibling, parent, or other more remote ancestor. It is required that **level** > 0. There must already exist grids marked as belonging to refinement level **level** $- 1$. The **indexRange()** (§3.2.3) of the refinement is computed from **range**, which must be a subset of the index range of a refinement of the same base grid (the refinement grid's most remote ancestor), at the next-coarser level of refinement (the refinement level of any parent grid), that would cover the entire base grid. The refinement factor **factor** is relative to any parent grid; any cell of any parent grid that is covered by cells of the new refinement grid is subdivided in each direction k into **factor**(k) cells of the new refinement grid. This function returns the index of the refinement grid.

6.1.16 Integer addRefinement(const IntegerArray& range, const Integer& factor, const Integer& level, const Integer k = 0)

Add a refinement grid to this collection. This function simply calls **addRefinement()** (§6.1.15) with **factor** replaced by an **IntegerArray** of length three set to the scalar **factor**. This function returns the index of the refinement grid.

6.1.17 virtual void deleteRefinement(const Integer& k)

Delete all multigrid levels of refinement grid **k**.

6.1.18 virtual void deleteRefinementLevels(const Integer& level)

Delete all grids in this collection marked as belonging to refinement levels greater than **level**. It is required that **level** ≥ 0.

6.1.19 void referenceRefinementLevels(CompositeGrid& x, Integer level = INTEGER_MAX)

Make this collection contain exactly those grids from the collection **x** which are marked in **x** as belonging to refinement level **level** or to any coarser refinent level. It is required that **level** ≥ 0.

6.1.20 virtual Integer addMultigridCoarsening(const IntegerArray& factor, const Integer& level, const Integer k = 0)

Add a multigrid coarsening of a grid to this collection. The multigrid coarsening factor in each index direction, relative to the next-finer multigrid level, is given by **factor**. This coarsening is marked as belonging to multigrid level **level**. It is marked as belonging to the same component grid as that of **grid**[**k**], which may be any finer multigrid level of the same component grid. It is required that **level** > 0. There must already exist a multigrid coarsening of the component grid at multigrid level **level** $- 1$. This function returns the index of the multigrid coarsening.

6.1.21 Integer addMultigridCoarsening(const Integer& factor, const Integer& level, const Integer k = 0)

Add a multigrid coarsening of a grid to this collection. The multigrid coarsening factor relative to the next-finer multigrid level is given by **factor**. This coarsening is marked as belonging to multigrid level **level**. It is marked as belonging to the same component grid as that of **grid**[**k**], which may be any finer multigrid level of the same component grid. It is required that **level** > 0. There must already exist a multigrid coarsening of the component grid at multigrid level **level** $- 1$. This function returns the index of the multigrid coarsening.

6.1.22 void makeCompleteMultigridLevels()

Add multigrid coarsenings of component grids as needed in order to complete the multigrid levels, so that each component grid, including any and all refinement grids, has at least **numberOfCompleteMultigridLevels**() multigrid levels. There should already exist at least two multigrid levels of each unrefined component grid. Otherwise, any multigrid levels added will use default values for multigrid parameters, which may be inappropriate. Any multigrid levels added to unrefined component grids that already have at least one but fewer than **numberOfCompleteMultigridLevels**() multigrid levels take their multigrid parameters from the next-finer multigrid level of the same component grid. Any multigrid levels added to refinement grids take their multigrid parameters from the same multigrid level of the corresponding unrefined component grid.

6.1.23 virtual void deleteMultigridCoarsening(const Integer& k)

Delete grid **k**,a multigrid coarsening, and all of its coarser multigrid levels.

6.1.24 virtual void deleteMultigridLevels(const Integer level = 0)

Delete all grids in this collection marked as belonging to multigrid levels greater than **level**. It is required that **level** ≥ 0.

6.1.25 void getInterpolationStencil(const Integer& k1, const IntegerArray& k2, const RealArray& r, IntegerArray& interpolationStencil, const LogicalArray& useBackupRules)

g (INPUT) The unrefined grid corresponding to grid **k2**.

k1 (INPUT) The index of the grid containing the interpolation points.

k2 (INPUT) The index of the grid containing the interpolee points.

r (INPUT) Dimensions: $(N, 0{:}\, n_1)$
The interpolation coordinates of the interpolation points. The range of points N for which the boundary is adjusted is determined by the first dimension of **r**. The first dimension of **interpolationStencil** and **useBackupRules** should be the same as the first the dimension of **r**. The second dimension of **r** and the third dimension of **interpolationStencil** should have $n_1 \geq$ **numberOfDimensions**$() - 1$.

interpolationStencil (OUTPUT) Dimensions: $(N, 0{:}\, 2, 0{:}\, n_1)$
The ranges of indices of interpolee points on grid **k2**.

useBackupRules (INPUT) Dimensions: (N)
An array of flags to indicate which points use backup interpolation rules.

This function computes the index bounds on stencils of points used for interpolation. If grid **k1** is a refinement grid, then the interpolation stencil may contain points from more then one component grid. In particular, it may contain points from grid **k2** and/or any of its sibling component grids (refinements of the same base grid at the same refinement level). For this case, it is necessary to call the variant of **getInterpolationStencil**() which takes the base grid as an argument.

6.2 Public Member functions for access to data

6.2.1 Integer& numberOfCompleteMultigridLevels()

This function returns a reference to the number of complete multigrid levels, the number of multigrid levels at which a complete set of component grids exists. Interpolation is in general possible on complete multigrid levels.

6.2.2 Real& epsilon()

This function returns a reference to the tolerance used in computing interpolation stencils.

6.2.3 Logical& interpolationIsAllExplicit()

This function returns a reference to a flag that indicates whether all interpolation is guaranteed to be explicit. This flag is always recomputed by the function **update(what,how)** (§6.1.12).

6.2.4 Logical& interpolationIsAllImplicit()

This function returns a reference to a flag that indicates whether all interpolation is allowed to be implicit. This flag is always recomputed by the function **update(what,how)** (§6.1.12).

6.2.5 virtual aString getClassName() const

Get the class name of the most-derived class for this object.

6.2.6 CompositeGridData$*$ operator$->$()

Access the reference-counted data.

6.2.7 CompositeGridData& operator$*$()
const CompositeGridData& operator$*$() const

Access the pointer to the reference-counted data.

6.3 Public member functions called only from derived classes

It is recommended that these functions be called only from derived classes.

6.3.1 void reference(CompositeGridData& x)

Make a reference to an object of type **CompositeGridData**. This **CompositeGrid** uses **x** for its data. It is recommended that this function be called only from derived classes.

6.3.2 void updateReferences(const Integer what = EVERYTHING)

Update references to the reference-counted data. It is recommended that this function be called only from derived classes.

6.3.3 virtual void setNumberOfGrids(const Integer& numberOfGrids_)

Set the number of grids, and add or delete grids to/from this collection until there are **numberOfGrids_** grids. See also **setNumberOfDimensionsAndGrids(numberOfDimensions_, numberOfGrids_)** (§6.3.5). It is recommended that this function be called only from derived classes.

6.3.4 virtual void setNumberOfDimensions(const Integer& numberOfDimensions_)

Set the number of dimensions. This is valid only if the collection does not contain **MappedGrid**s with a different number of dimensions. See also **setNumberOfDimensionsAndGrids(numberOfDimensions_, numberOfGrids_)** (§6.3.5). It is recommended that this function be called only from derived classes.

6.3.5 virtual void setNumberOfDimensionsAndGrids(const Integer& numberOfDimensios_, const Integer& numberOfGrids_)

Set the number of dimensions of the grids, and add or delete grids to/from this collection until there are **numberOfGrids_** grids. Any grids added are marked as belonging to base grid zero, multigrid level zero and refinement level zero. The use of this function is not recommended if the collection is intended to contain more than one base grid, multigrid level or refinement level. For example, if the collection is intended to contain more than one refinement level, it is recommended that the functions **addRefinement(range,factor,level,k)** (§6.1.15) and **deleteRefinementLevels(level)** (§6.1.18) be used instead. It is recommended that this function be called only from derived classes.

6.4 Public data

6.4.1 IntegerArray numberOfInterpolationPoints

Dimensions: $(0{:}\,2, 0{:}\,$**numberOfGrids**$() - 1)$
The number of interpolation points on each component grid.

6.4.2 IntegerArray numberOfInterpoleePoints

Dimensions: $(0{:}\,2, 0{:}\,$**numberOfGrids**$() - 1)$
The number of interpolee stencil points on each component grid, in stencils that contain points from more than one interpolee grid.

6.4.3 LogicalArray interpolationIsImplicit

Dimensions: $(0{:}\,2,\ 0{:}\,$**numberOfComponentGrids**$() - 1,\ 0{:}\,$**numberOfComponentGrids**$() - 1,\ 0{:}\,$**numberOfMultigridLevels**$() - 1)$
The type of interpolation (to-grid, from-grid, multigrid-level).

6.4.4 LogicalArray backupInterpolationIsImplicit

Dimensions: $(0{:}\,2,\ 0{:}\,$**numberOfComponentGrids**$() - 1,\ 0{:}\,$**numberOfComponentGrids**$() - 1,\ 0{:}\,$**numberOfMultigridLevels**$() - 1)$
The type of interpolation (to-grid, from-grid, multigrid-level) when using backup interpolation rules.

6.4.5 IntegerArray interpolationWidth

Dimensions: $(0{:}\,2,\ 0{:}\,$**numberOfComponentGrids**$() - 1,\ 0{:}\,$**numberOfComponentGrids**$() - 1,\ 0{:}\,$**numberOfMultigridLevels**$() - 1)$
The width of the interpolation stencil (direction, to-grid, from-grid, multigrid-level).

6.4.6 IntegerArray backupInterpolationWidth

Dimensions: $(0{:}\,2,\quad 0{:}\,$**numberOfComponentGrids**$() - 1,\quad 0{:}\,$**numberOfComponentGrids**$() - 1,\quad 0{:}\,$**numberOfMultigridLevels**$() - 1)$
The width of the interpolation stencil (direction, to-grid, from-grid, multigrid-level) when using backup interpolation rules.

6.4.7 RealArray interpolationOverlap

Dimensions: $(0{:}\,2,\quad 0{:}\,$**numberOfComponentGrids**$() - 1,\quad 0{:}\,$**numberOfComponentGrids**$() - 1,\quad 0{:}\,$**numberOfMultigridLevels**$() - 1)$
The minimum overlap for interpolation (direction, to-grid, from-grid, multigrid-level).

6.4.8 RealArray backupInterpolationOverlap

Dimensions: $(0{:}\,2,\quad 0{:}\,$**numberOfComponentGrids**$() - 1,\quad 0{:}\,$**numberOfComponentGrids**$() - 1,\quad 0{:}\,$**numberOfMultigridLevels**$() - 1)$
The minimum overlap for interpolation (direction, to-grid, from-grid, multigrid-level) when using backup interpolation rules.

6.4.9 RealArray interpolationConditionLimit

Dimensions: $(0{:}\,2,\quad 0{:}\,$**numberOfComponentGrids**$() - 1,\quad 0{:}\,$**numberOfComponentGrids**$() - 1,\quad 0{:}\,$**numberOfMultigridLevels**$() - 1)$
The maximum interpolation condition number allowed for interpolation (to-grid, from-grid, multigrid-level).

6.4.10 RealArray backupInterpolationConditionLimit

Dimensions: $(0{:}\,2,\quad 0{:}\,$**numberOfComponentGrids**$() - 1,\quad 0{:}\,$**numberOfComponentGrids**$() - 1,\quad 0{:}\,$**numberOfMultigridLevels**$() - 1)$
The maximum interpolation condition number allowed for interpolation (to-grid, from-grid, multigrid-level) when using backup interpolation rules.

6.4.11 LogicalArray interpolationPreference

Dimensions: $(0{:}\,2,\quad 0{:}\,$**numberOfComponentGrids**$() - 1,\quad 0{:}\,$**numberOfComponentGrids**$() - 1,\quad 0{:}\,$**numberOfMultigridLevels**$() - 1)$
A list of the indices of component grids from which each component grid may interpolate (list, to-grid, multigrid-level), in increasing order of preference. Discretization must be included in this list, and is indicated by the component-grid index "to-grid." The list is terminated by -1 if there are fewer than **numberOfComponentGrids()** grids in the list.

6.4.12 LogicalArray mayInterpolate

Dimensions: $(0{:}\,$**numberOfComponentGrids**$() - 1,\ 0{:}\,$**numberOfComponentGrids**$() - 1,\ 0{:}\,$**numberOfMultigridLevels**$() - 1)$
Flags indicating which component grids may interpolate from each other (to-grid, from-grid, multigrid-level).

6.4.13 LogicalArray mayBackupInterpolate

Dimensions: $(0{:}\,2,\quad 0{:}\,$**numberOfComponentGrids**$() - 1,\quad 0{:}\,$**numberOfComponentGrids**$() - 1,\quad 0{:}\,$**numberOfMultigridLevels**$() - 1)$
Flags indicating which grids may interpolate from each other (to-grid, from-grid, multigrid-level) when using backup interpolation rules.

6.4.14 LogicalArray mayCutHoles

Dimensions: $(0{:}\,2,\ 0{:}\,$**numberOfComponentGrids**$() - 1, 0{:}\,$**numberOfComponentGrids**$() - 1)$
Flags indicating which component grids may cut holes in each other (cutting-grid, cut-grid, multigrid-level), where the boundary of one grid intersects the interior of another grid and that boundary is labelled as a domain boundary. Normally, all sides of grids labelled as domain boundaries are used to cut holes in all other grids.

6.4.15 LogicalArray multigridCoarseningRatio

Dimensions: $(0{:}\,2,\ 0{:}\,$**numberOfComponentGrids**$() - 1, 0{:}\,$**numberOfMultigridLevels**$() - 1)$
multigridCoarseningRatio holds the ratio in each index direction, for each component grid, of the size of cells of that grid to the size of cells of the next-finer multigrid level of the same component grid.

6.4.16 LogicalArray multigridProlongationWidth

Dimensions: $(0{:}\,2,\ 0{:}\,$**numberOfComponentGrids**$() - 1, 0{:}\,$**numberOfMultigridLevels**$() - 1)$
multigridProlongationWidth holds the stencil width in each index direction, for each component grid for the multigrid prolongation operator used to transfer data to the next-finer multigrid level of the same component grid. For the finest multigrid level (level zero), this parameter has no meaning.

6.4.17 LogicalArray multigridRestrictionWidth

Dimensions: $(0{:}\,2,\ 0{:}\,$**numberOfComponentGrids**$() - 1, 0{:}\,$**numberOfMultigridLevels**$() - 1)$
multigridRestrictionWidth holds the stencil width in each index direction, for each component grid for the multigrid restriction operator used to transfer data to the next-coarser multigrid level of the same component grid. For the coarsest multigrid level, this parameter has no meaning.

6.4.18 ListOfRealArray interpolationCoordinates

Length: **numberOfGrids()** (§4.2.2)
Dimensions of **interpolationCoordinates**[k]: $(0{:}\,$**numberOfInterpolationPoints**$(k) - 1,\ 0{:}\,$**numberOfDimensions**$() - 1)$
interpolationCoordinates holds the coordinates of each interpolation point in the parameter space of its interpolee grid.

6.4.19 ListOfIntegerArray interpoleeGrid

Length: **numberOfGrids()** (§4.2.2)
Dimensions of **interpoleeGrid**[k]: $(0{:}\,$**numberOfInterpolationPoints**$(k) - 1)$
interpoleeGrid holds the component-grid number of the interpolee grid of each interpolation point. If the interpolation stencil for interpolation point j contains points from more than one interpolee grid, then the expression **interpoleeGrid**$[k](j)\&$**CompositeGrid**$::$**ISgivenByInterpoleePoint** is non-zero, and the expression **interpoleeGrid**$[k](j)\&$**CompositeGrid**$::$**GRIDnumberBits** gives the nominal interpolee grid number. In that case, the interpolation coordinates should be interpreted in the context of the nominal interpolee grid. It may happen that the interpolation coordinates lie outside the interval $[0, 1]$. This is not an error, but indicates only that the interpolation point lies outside the nominal interpolee grid even though it lies within the stencil of interpolee points on their various respective grids. For each interpolation stencil that contain points from more than one interpolee grid, the interpolee component-grid number of each point in the interpolation stencil is stored in **interpoleePoint**$[k]$ (§6.4.21).

6.4.20 IntegerArray interpoleeGridRange

Dimensions: $(0{:}\,$**numberOfComponentGrids**$(),\ 0{:}\,$**numberOfComponentGrids**$() - 1,\ 0{:}\,$**numberOfMultigridLevels**$() - 1)$
The starting index within **interpoleeGrid**$[k]$, where grid k is component grid k_2 at multigrid level l, for points interpolated from component grid k_1 is **interpoleeGridRange**(k_1, k_2, l), and the ending index is **interpoleeGridRange**$(k_1 + 1, k_2, l) - 1$. For all indices i within this range, **interpoleeGrid**$[k](i) = k_2$. These index ranges do not include the indices of any points whose interpolation stencils contain interpolee points from more than one grid. However, as these are always the last interpolation points listed, it is not hard to see that the starting index of such points is **interpoleeGridRange**(**numberOfComponentGrids**$, k_2, l)$, and the ending index is **numberOfInterpolationPoints**$(k_2) - 1$.

6.4.21 ListOfIntegerArray interpoleePoint

Length: **numberOfGrids()** (§4.2.2)
Dimensions of **interpoleePoint**[k]: $(0{:}\,$**numberOfInterpoleePoints**$(k) - 1)$
interpoleePoint holds the component-grid number of the interpolee grid of each interpolee point in those interpolation stencils which contain points from more than one interpolee grid. These component-grid numbers are listed in order of increasing $i1$, then increasing $i2$, and then increasing $i3$, where $(i1, i2, i3)$ are the indices of the interpolee points in their stencil, and finally, in the order that the multiple-interpolee-grid interpolation stencils are encountered in **interpoleeGrid**$[k]$ (§6.4.19).

6.4.22 ListOfIntegerArray interpoleeLocation

Length: **numberOfGrids()** (§4.2.2)
Dimensions of **interpoleeLocation**[k]: $(0\colon$ **numberOfInterpolationPoints**$(k)-1,\ 0\colon$ **numberOfDimensions**$()-1)$
interpoleeLocation holds the indices (in the interpolee grid) of the lower-left corner of the interpolation stencil for each interpolation point.

6.4.23 ListOfIntegerArray interpolationPoint

Length: **numberOfGrids()** (§4.2.2)
Dimensions of **interpolationPoint**[k]: $(0\colon$ **numberOfInterpolationPoints**$(k)-1,\ 0\colon$ **numberOfDimensions**$()-1)$
interpolationPoint holds the indices (in the grid of the interpolation point) of each interpolation point.

6.4.24 ListOfRealArray interpolationCondition

Length: **numberOfGrids()** (§4.2.2)
Dimensions of **interpolationCondition**[k]: $(0\colon$ **numberOfInterpolationPoints**$(k)-1)$
interpolationCondition holds the condition number for interpolation of each interpolation point.

6.4.25 ListOfCompositeGrid multigridLevel

multigridLevel is a list of **CompositeGrid**s containing those **MappedGrid**s which belong to the same multigrid level. **multigridLevel** overloads **GridCollection::multigridLevel** (§5.4.8), whose **GridCollection**s contain the same grids as the **CompositeGrid**s of **multigridLevel**. This data may be updated as in the following example.

6.5 Public data used by the Grid Generator Ogen

Data used by class Ogen for optimization of overlap computation.

6.5.1 RealCompositeGridFunction inverseCondition

Dimensions of **inverseCondition**[k]: $(d_{00}\colon d_{10}, d_{01}\colon d_{11}, d_{02}\colon d_{12})$, where $d_{ij} =$ **grid**[k].**dimension**(i,j).
inverseCondition holds the condition number for interpolation of each point, in case that point were to be interpolated from the grid whose index is given by **inverseGrid** (§6.5.3).

6.5.2 RealCompositeGridFunction inverseCoordinates

Dimensions of **inverseCoordinates**[k]: $(d_{00}\colon d_{10}, d_{01}\colon d_{11}, d_{02}\colon d_{12}, 0\colon n_1)$, where $d_{ij} =$ **grid**[k].**dimension**(i,j) and $n_1 =$ **numberOfDimensions**$()-1$.
inverseCoordinates holds the coordinates for interpolation of each point, in case that point were to be interpolated from the grid whose index is given by **inverseGrid** (§6.5.3).

6.5.3 IntegerCompositeGridFunction inverseGrid

Dimensions of **inverseGrid**[k]: $(d_{00}\colon d_{10}, d_{01}\colon d_{11}, d_{02}\colon d_{12})$, where $d_{ij} =$ **grid**[k].**dimension**(i,j).
inverseGrid holds, for each point, the index of a grid for which interpolation coordinates have been computed, in case that point were to be interpolated from that grid. The index -1 is used to indicate the case where interpolation coordinates have not been or cannot be computed.

6.6 Public data used only by derived classes

It is recommended that these variables be used only by derived classes.

6.6.1 CompositeGridData* rcData

rcData is a pointer to the reference-counted data. It is recommended that this variable be used only by derived classes. See also the member functions **operator->()** (§6.2.6) and **operator*()** (§6.2.7), which are provided for access to **rcData**.

6.6.2 Logical isCounted

isCounted is a flag that indicates whether the data pointed to by **rcData** (§6.6.1) is known to be reference-counted. It is recommended that this variable be used only by derived classes.

6.7 Public constants

6.7.1 THEinterpolationCoordinates

THEinterpolationCoordinates indicates **interpolationCoordinates** (§6.4.18), the coordinates of the inteprolation points in the parameter space of the grids from which they interpolate. See also **update(what,how)** (§6.1.12) and **destroy(what)** (§6.1.14).

6.7.2 THEinterpoleeGrid

THEinterpoleeGrid indicates **interpoleeGrid** (§6.4.19), the indices of the grids from which the inteprolation points interpolate. See also **update(what,how)** (§6.1.12) and **destroy(what)** (§6.1.14).

6.7.3 THEinterpoleeLocation

THEinterpoleeLocation indicates **interpoleeLocation** (§6.4.22), the indices of the lower-left corners of the interpolation stencils in the grids from which the inteprolation points interpolate. See also **update(what,how)** (§6.1.12) and **destroy(what)** (§6.1.14).

6.7.4 THEinterpolationPoint

THEinterpolationPoint indicates **interpolationPoint** (§6.4.23), the indices of the interpolation points. See also **update(what,how)** (§6.1.12) and **destroy(what)** (§6.1.14).

6.7.5 THEinterpolationCondition

THEinterpolationCondition indicates **interpolationCondition** (§6.4.24), the condition numbers for interpolation of the interpolation points. See also **update(what,how)** (§6.1.12) and **destroy(what)** (§6.1.14).

6.7.6 THEinverseMap

THEinverseMap indicates **inverseCondition** (§6.5.1), **inverseCoordinates** (§6.5.2) and **inverseGrid** (§6.5.3). See also **update(what,how)** (§6.1.12) and **destroy(what)** (§6.1.14).

6.7.7 THEusualSuspects

THEusualSuspects = **GridCollection::THEusualSuspects** (§5.6.21) | **THEinterpolationCoordinates** (§6.7.1) | **THEinterpoleeGrid** (§6.7.2) | **THEinterpoleeLocation** (§6.7.3) | **THEinterpolationPoint** (§6.7.4)
See also **update(what,how)** (§6.1.12) and **destroy(what)** (§6.1.14).

6.7.8 EVERYTHING

EVERYTHING = **GridCollection::EVERYTHING** (§5.6.22) | **THEinterpolationCoordinates** (§6.7.1) | **THEinterpoleeGrid** (§6.7.2) | **THEinterpoleeLocation** (§6.7.3) | **THEinterpolationPoint** (§6.7.4) | **THEinterpolationCondition** (§6.7.5) | **THEinverseMap** (§6.7.6)
See also **update(what,how)** (§6.1.12) and **destroy(what)** (§6.1.14).

6.7.9 COMPUTEtheUsual

COMPUTEtheUsual = **GridCollection::COMPUTEtheUsual** (§5.6.26)
See also **update(what,how)** (§6.1.12) and **destroy(what)** (§6.1.14).

6.7.10 ISgivenByInterpoleePoint

ISgivenByInterpoleePoint is used with **interpoleeGrid**$[k]$ (§6.4.19) to indicate interpolation points whose interpolation stencils contain points from more than one interpolee grid.

A Class ReferenceCounting

A.1 Public member functions

A.1.1 ReferenceCounting()

Default constructor.

A.1.2 ReferenceCounting(const ReferenceCounting& x, const CopyType ct = DEEP)

Copy constructor. This does a deep copy by default. In fact, for class **ReferenceCounting**, there is no data to be copied, so the arguments x and ct are ignored.

A.1.3 virtual ∼ReferenceCounting()

Destructor. This function checks that the actual number of references is zero, as it should be when the object is destroyed.

A.1.4 virtual ReferenceCounting& operator=(const ReferenceCounting& x)

Assignment operator. This is also called a deep copy.

A.1.5 virtual void reference(const ReferenceCounting& x)

Make a reference. This is also called a shallow copy. This **ReferenceCounting** shares the data of **x**. In fact, this virtual function does nothing at all in the base class **ReferenceCounting**, but it should have the effect specified above when it is defined in any envelope class that is derived from **ReferenceCounting**.

A.1.6 virtual void breakReference()

Break a reference. If this **ReferenceCounting** shares data with any other **ReferenceCounting**, then this function replaces it with a new copy that does not share data. In fact, this virtual function does nothing at all in the base class **ReferenceCounting**, but it should have the effect specified above when it is defined in any envelope class that is derived from **ReferenceCounting**.

A.1.7 virtual ReferenceCounting∼ virtualConstructor(const CopyType ct = DEEP) const

Allocate and return a pointer to a new copy of this **ReferenceCounting**. Any class that is derived from **ReferenceCounting** should also define this virtual function, so that this function will allocate a new copy of an object of the derived class.

A.1.8 Integer incrementReferenceCount()

Increment the reference count and return the resulting value.

A.1.9 Integer decrementReferenceCount()

Decrement the reference count and return the resulting value.

A.1.10 Logical uncountedReferencesMayExist()

This function determines whether uncounted references may exist. If so, it returns a non-zero value; otherwise it returns zero. If it is determined that uncounted references may exist, then **getReferenceCount()** (§A.2.1) thereafter returns one more than the actual number of references. This is done so that when all of the counted references are destroyed and the actual number of references is decremented to zero, **getReferenceCount()** will still return the value one, to indicate that one or more uncounted references to the object may still exist and that the object should still not be deleted.

A.1.11 virtual void consistencyCheck() const

Check the consistency of this **ReferenceCounting**.

A.2 Public Member functions for access to data

A.2.1 Integer getReferenceCount()

This function returns the reference count.

A.2.2 virtual aString getClassName() const

Get the class name of the most-derived class for this object.

A.2.3 Integer getGlobalID() const

Get the unique global identifier for this **ReferenceCounting**.

B Stream I/O

B.1 Stream I/O Operators

B.1.1 ostream& operator≪(ostream& s, const ReferenceCounting& x)

Stream output operator.

B.1.2 ostream& operator≪(ostream& s, const GenericGrid& g)

Stream output operator.

B.1.3 ostream& operator≪(ostream& s, const MappedGrid& g)

Stream output operator.

B.1.4 ostream& operator≪(ostream& s, const GenericGridCollection& g)

Stream output operator.

B.1.5 ostream& operator≪(ostream& s, const GridCollection& g)

Stream output operator.

B.1.6 ostream& operator≪(ostream& s, const CompositeGrid& g)

Stream output operator.

Index